LAC2014

12th Linux Audio Conference

The Open Source Music and Sound Conference

PROCEEDINGS

01.–04.05.2014

ZKM | KARLSRUHE

Institute for Music and Acoustics

lac.linuxaudio.org/2014

zkm karlsruhe

CiTu

PARAGRAPHE UNIVERSITÉ PARIS 8

linux audio .org

Published by
Institute for Music and Acoustics (IMA),
ZKM | Center for Art and Media Karlsruhe, Germany
May 2014
Editors: Götz Dipper, Robin Gareus, Frank Neumann and Jochen Arne Otto

`http://lac.linuxaudio.org/2014`
ISBN 978-3-928201-45-2

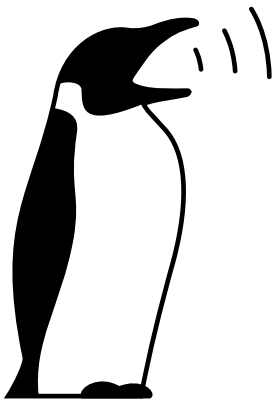
Credits
Cover Design: Nina Rüb
Layout and Typesetting: Frank Neumann and Jochen Arne Otto
with LaTeX and pdfLaTeX
Logo Design: The Linuxaudio.org logo and its variations ©Thorsten Wilms 2006,
imported into "LAC 2010" logo by Robin Gareus

Thanks to:
Martin Monperrus for his webpage "Creating proceedings from PDF files"

Printed in Germany

Partners and Sponsors

zkm karlsruhe



CiTu

PARAGRAPHE UNIVERSITÉ PARIS 8

Foreword

Welcome to the LAC2014 in Karlsruhe!

Around eleven years ago the Advanced Linux Sound Architecture (ALSA) was merged into the mainline Linux Kernel. It was introduced in the 2.5 development kernel and marked a major milestone in the GNU/Linux audio world.

In the wake of this, and after a small audio-focused event at the LinuxTag, a group of developers teamed up with enthusiasts with the goal to organize the first Linux Audio Conference: An event to exchange knowledge and foster discussions surrounding Linux kernel-based systems and allied libre software for audio-related work.

The LAC was born and its first instalment was hosted at the ZKM in March 2003.

Due to its success it became an annual event with the first four conferences 2003–2006 taking place at the ZKM after which the conference started traveling, first nationally, then internationally, evolving and growing with each iteration.

Fast-forward 11 years in time, we can say without false modesty that the endeavour thrived. It grew out of a small group of developers into a conference attracting artists and academics alike. The number of projects that were initially conceived at LACs over the years only underlines the importance of personal face-to-face meetings or discussions which is facilitated by a conference like the LAC.
While many things have changed since, we are pleased that the fundamentals of the LAC laid down by the first organizers have survived, in particular scientific rigour. As opposed to many other similar Linux developer or hacker events, the LAC is one of the few that retains a peer-review, and these printed proceedings emphasize the importance of written documentation, particularly in a field where interface definitions as well as signal processing mathematics are important.

As for new additions, LAC'14 will pick up on last year's schedule which featured a well received day-off with an excursion to the countryside. Sunday, 4 May this year is reserved for a trip to the nearby city of Bruchsal and its Château, which hosts the German Museum of Mechanical Musical Instruments.

Similarly the dedicated slot for lightning talks reappears on the schedule, an opportunity for developers to pitch ongoing projects and present their works in a less formal session. A new addition to the LAC'14 program is the poster session, which we are confident to prove useful to complement the paper presentations taking place over the course of three days.

We are excited about the Linux Audio Conference 2014, featuring a tightly packed, diverse schedule with 77 events by over 100 persons! Five full evening concerts and 25 presentations in just three days. Even though the schedule is tightly packed, it was a tough decision for the music jury and paper review committee to choose from an even

greater number of excellent submissions taking into account the inevitable limitations in time and resources.

The 12th anniversary this year marks the 'LAC coming home' to ZKM. We hope to provide a pleasant experience to all conference participants and wish that you feel at home here, as well.

We would like to express our gratitude to everyone involved with the conference and particularly like to thank Linus van Geuns for kindly lending us his video streaming server hardware.

Most importantly we would like to thank you, the worldwide Linux Audio Community. Neither the conference nor Linux would work if it was not for the presence of such a dedicated group.

Have a great time in Karlsruhe!

Ludger Brümmer, Götz Dipper, Robin Gareus,
Frank Neumann and Jochen Arne Otto

Conference Organization Core Team

Götz Dipper
Robin Gareus
Frank Neumann
Jochen Arne Otto

Conference Website and Design

Robin Gareus
Nina Rüb

Paper Administration and Proceedings

Frank Neumann

Organization of Music Programme, Installations, and Workshops

Michael Hohendorf
Marie-Kristin Meier

ZKM | Institute for Music and Acoustics

Ludger Brümmer (head)
Götz Dipper
Alexander Hofmann
Michael Hohendorf
Yana Il
Anton Kossjanenko
Mara May
Marie-Kristin Meier
Marcel Mendel
Caro Mössner
Matthias Müller
Jochen Arne Otto
Sebastian Schottke
Holger Stenschke
Bernhard Sturm
David Wagner

ZKM | Events

Hartmut Bruckner
Hans Gass
Berthold Schwarz
Florian Vitez
Manuel Weber

ZKM | IT, Video Documentation and Stream Team

Moritz Büchner	Martina Rotzal
Linus van Geuns	Joachim Schütze
Frank Neumann	Fabian Selbach
Paula Reissig	Christina Zartmann

Special Thanks

Marc Groenewegen	Benny Lyons
Helene Hedsund	Koen Pepping
Frits van der Holst	Casper Ravenhorst
Bass Jansson	Funs Seelen
Magnus Johansson	Pieter Suurmond
Pjotr Lasschuit	Christian Thäter
Björn Lindig	Hermann Voßeler

Public Relations, Marketing, and Online Communication

Dominika Szope	Verena Noack
Julia Wicky	Tim Welker
Siemke Hanßen	Wera Schnürer
Constanze Heidt	

... and thanks to everyone else who helped in numerous places after the editorial deadline of this publication.

Review Committee

Fons Adriaensen	Casa della Musica, Parma, Italy
Marije Baalman	nescivi / STEIM, Netherlands
Tim Blechmann	Austria
Ivica Bukvic	Virginia Tech, United States
Götz Dipper	ZKM | Institut für Musik und Akustik, Germany
John ffitch	United Kingdom
Robin Gareus	linuxaudio.org, University Paris 8, France
Albert Gräf	Johannes Gutenberg University Mainz, Germany
Marc Groenewegen	Utrecht School of Music & Technology, Netherlands
Daniel James	64 Studio Ltd., United Kingdom
Victor Lazzarini	NUIM Maynooth, Ireland
Björn Lindig	Germany
Fernando Lopez-Lezcano	CCRMA, Stanford University, United States
Jörn Nettingsmeier	Freelancer, Germany
Yann Orlarey	Grame - Centre National de Création Musicale, France
Jochen Arne Otto	ZKM | Institut für Musik und Akustik, Germany
Martin Rumori	Institute of Music and Performing Arts Graz, Austria
Bruno Ruviaro	Santa Clara University, United States
Funs Seelen	[muditulib], Netherlands
Pieter Suurmond	HKU University of the Arts Utrecht, Netherlands
IOhannes m zmölnig	Institute of Electronic Music and Acoustics Graz, Austria

Music Jury

Ludger Brümmer	ZKM | Institut für Musik und Akustik, Germany
Götz Dipper	ZKM | Institut für Musik und Akustik, Germany
Robin Gareus	linuxaudio.org, University Paris 8, France
Michael Hohendorf	ZKM | Institut für Musik und Akustik, Germany
Yana Il	ZKM | Institut für Musik und Akustik, Germany
Paul Modler	Hochschule für Gestaltung Karlsruhe, Germany
Jochen Arne Otto	ZKM | Institut für Musik und Akustik, Germany

Table of Contents

A TouchOSC MIDI Bridge for Linux

Albert GRÄF
Computer Music Research Group
Institute of Art History and Musicology (IKM)
Johannes Gutenberg University (JGU) Mainz, Germany
aggraef@gmail.com

Abstract

Mobile applications such as hexler's TouchOSC offer a cheap and convenient alternative to traditional controller hardware for computer music programs. TouchOSC is available for Android and iOS devices and supports both OSC and MIDI, two widespread standards for transmitting control data between computer music applications. On the host side the TouchOSC MIDI Bridge is required for MIDI support, which unfortunately is proprietary software and only available for Mac and Windows systems. This paper presents pd-touchosc, a library of Pd externals which aims to bring most of the functionality of the TouchOSC MIDI Bridge to Linux.

Keywords

TouchOSC, controller, OSC, MIDI

1 Introduction

Any reader familiar with the area of computer music will have heard of JazzMutant's Lemur controller [5], a big multitouch device with built-in OSC[1] and MIDI[2] support, which was fully configurable using a kind of GUI builder for control surfaces. Nowadays, the Lemur's place is taken by mobile apps running on modern (and much cheaper) devices such as smartphones and tablets. (It is no accident that the demise of the original Lemur hardware was brought about by the advent of the iPad.) The Lemur lives on as a mobile app on iOS[3], and there are other similar apps on both Android and iOS.

One of these is hexler's TouchOSC [4]. While it lacks some of the Lemur's more advanced features such as physical models and scripting capabilities, it certainly offers enough features to create fairly sophisticated interfaces and is also much cheaper. It comes with its own graphical layout editor (which is written in Java and thus runs on Linux just as well as on Mac and Windows). Like the Lemur, TouchOSC supports both OSC and MIDI and the layout of the controller elements is fully configurable, so that the user can tailor the graphical interface to the computer music application at hand. This sets it apart from applications like TouchDAW[4] which provide fixed interfaces usually inspired by existing MIDI controller designs. There are other apps similar to the Lemur and TouchOSC, such as OSCPad[5] which is more or less compatible with the TouchOSC layout format, and Charlie Roberts' open-source app Control[6] which features its own JSON format and is both scriptable and extensible using JavaScript. But among these TouchOSC seems to be the most mature and popular option right now, not least because of its graphical layout editor.

One of the downsides of TouchOSC for Linux users, however, is its MIDI support which requires either the TouchOSC MIDI Bridge program or an RTP-MIDI[7] interface on the host side, neither of which is readily available on Linux. (The TouchOSC MIDI Bridge is closed source software only available for Mac and Windows, and drivers for RTP-MIDI are hard to find for Linux these days. Moreover, the RTP-MIDI protocol doesn't seem to be supported in the Android version of the TouchOSC app anyway.) So the author set out to create a TouchOSC MIDI bridge replacement for Linux, which is what this paper is about.

Why would you want to use MIDI with TouchOSC anyway? It is true that MIDI is a much more limited format for control data than OSC, but you may find the conversion from/to MIDI convenient when interfacing TouchOSC to existing MIDI applications and hardware, such as synthesizers, algorithmic composition and music notation software, as well as DAW

[1] http://opensoundcontrol.org/
[2] http://www.midi.org/
[3] http://liine.net/en/products/lemur/
[4] http://www.humatic.de/htools/touchdaw/
[5] http://burnsmod.com/software/oscpad.html
[6] http://charlie-roberts.com/Control/
[7] http://www.cs.berkeley.edu/~lazzaro/rtpmidi/

(digital audio workstation) and sequencer programs. In particular, the conversion enables you to record control and automation data with DAW and sequencer programs, which typically offer good facilities for recording, playing back and editing MIDI sequences, but often provide only limited support for OSC, if at all.

So there are plenty of use cases for TouchOSC MIDI on Linux. Given that neither the proprietary TouchOSC Bridge protocol nor RTP-MIDI will work for our purposes, the most straightforward solution is to just take the MIDI mapping information available in TouchOSC layout files and convert OSC messages to/from MIDI in an automatic fashion using that information. This approach obviously has some shortcomings when compared to the "official" TouchOSC MIDI Bridge which connects directly to the TouchOSC app on the device. In particular, it requires a working OSC connection and that the TouchOSC layouts on the device and the host side match up. But this doesn't seem to be much of an impediment, and in any case it is better than not having any MIDI connectivity at all.

Our current implementation of the TouchOSC MIDI bridge for Linux uses Miller Puckette's Pd a.k.a. Pure Data[8], an interactive visual programming environment for computer music and multimedia applications. This makes it easy to create a working prototype of the software and also opens up the interface so that users can modify details of the implementation inside Pd. However, the core code of our solution (which is written in the author's Pure programming language [1]) could certainly be massaged into a stand-alone program which works outside the Pd environment.

2 TouchOSC Layouts

Let us begin with a brief overview of TouchOSC layouts. For further details we refer the reader to the documentation available at the TouchOSC website [4].

Layouts are created with the TouchOSC editor which can store them in zipped XML files or transfer them directly to a TouchOSC instance running on a device.

Figure 1 shows one page of a typical layout, as it is rendered on a device (an Android tablet in this case). When creating a layout with the editor, the user can choose from a built-in collection of various GUI widgets such as faders, rotary controls, push and toggle buttons and XY pads. These can be placed freely on the screen. A layout may consist of multiple pages which can be selected using the tabs at the top of the screen.

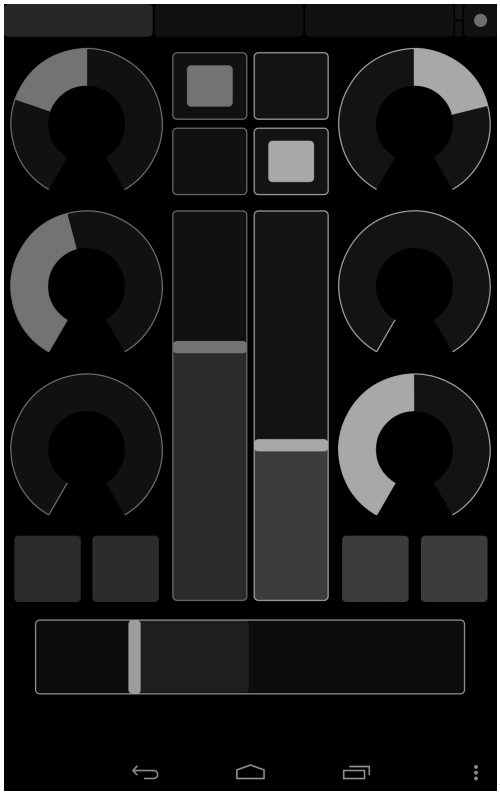


Figure 1: TouchOSC layout.

Each TouchOSC widget has one or more OSC messages associated with it, which are emitted when the status of the widget changes in some way (button pressed, fader moved, etc.). Conversely, OSC messages can also be transmitted to the device in order to change the current value of a widget. The following status variables are supported by most widgets:

- x, y: x is the primary value of a control, such as the value of a fader or a rotary, or the status of a button (0 = off, 1 = on). XY pads have a secondary y value, so they encode two values x and y (position of the control along the x and y axis, respectively) at the same time. These variables are for both input and output, i.e., they are transmitted to the host in response to a touch event, but the host can also send them back to the device in order to change the value. The latter is useful for setting up presets or providing visual feedback for some host-side operations on the device.

[8] `http://puredata.info/`

OSC message	Meaning
`/1`	first page
`/1/fader1 0.1`	value of `fader1` on the first page
`/1/fader1/color red`	color of `fader1` (input only)
`/1/fader1/z 1`	touch variable of `fader1` (output only)
`/1/xy1 0.1 0.7`	x, y values of a XY pad
`/1/multifader1/1 0.1`	value of the first subcontrol of a multi-fader
`/1/multifader1/1/z 1`	the subcontrol's z value
`/1/multixy1/1 0.1 0.7`	x, y values of the first subcontrol of a multi-XY pad
`/1/multipush1/2/3 0.1`	value of the subcontrol in column 2, row 3

Figure 2: TouchOSC message examples.

- z is the touch variable which can be either 1 if the widget is being touched or 0 otherwise. In the case of a touch button this will be the same as the primary value of the widget, but this value is output-only (it cannot be changed by transmitting a corresponding OSC message to the device).
- c is the color variable. TouchOSC offers a built-in palette of nine different colors which are usually set when editing the layout. But the color can also be changed dynamically by transmitting a corresponding OSC message to the device. This variable is input-only.

The OSC address of a widget can either be assigned automatically by the TouchOSC editor (in which case it takes the form */n/widget-name* where n is the number of the page on which the widget is located) or the user can set it manually to any valid OSC address string. This address is used for the primary widget value(s) (x and y), whereas the z and c values are specified by tacking on `/z` or `/color` to the OSC address. The OSC ranges of the numeric values are usually 0–1 by default, but this can be adjusted in the editor. The color variables have symbolic values such as `red`, `green` etc. in the OSC encoding.

Faders, buttons and XY pads also have multi-widget variations which consist of multiple controls of the same type making up a single widget. In this case the individual controls have separate OSC addresses of the form */widget-addr/i* with an index i ranging from 1 to the number of subcontrols, or (in the case of multi-button widgets) */widget-addr/i/j* where i denotes the column and j the row index (note that the column index comes first, even though TouchOSC arranges the subcontrols in row-major order internally).

Layout pages themselves also have an OSC address (by default, this will be simply `/1`, `/2`, etc.). A message with just the OSC address (without any parameters) will be emitted whenever the page is clicked in the tab strip, and the host can also send a message of this form to change the page that's currently displayed on the device.

Figure 2 summarizes the syntax of typical TouchOSC messages and their meaning.

3 MIDI Assignments

The TouchOSC editor allows MIDI messages to be assigned to any status variable of a widget in a layout. The details are a little intricate at first because of the distinct characteristics of the various widgets and the different kinds of MIDI messages, but work in rather intuitive and straightforward manner once the user is familiar with the available configuration options. TouchOSC supports all the different types of voice messages MIDI has on offer, as well as the sequencer-related system real-time messages (start, stop and continue).

The start, stop and continue messages offer no further configuration options. They can only be assigned to on/off variables, i.e., the primary value of buttons, or the touch value of any control. In our implementation, this kind of MIDI message is triggered whenever the corresponding control variable goes to a non-zero value.[9]

Voice messages generally map a control variable to the *last* data byte of the message. This will be the note velocity or control value for

[9] Note that, in contrast, the official TouchOSC MIDI Bridge seems to emit the message for each status change, i.e., also when the variable drops back to zero. We do not consider this behavior very useful, however, as it causes a sequencer message to be sent *twice* when pressing and releasing a push button. Nevertheless, there's a compilation time option in our code which provides compatibility with the TouchOSC MIDI Bridge in this respect if this is needed.

No.	Type	Channel	Fixed Data Value	Mapped Data Range
0	control change	1–16	controller number (0–127)	controller value (0–127)
1	note	1–16	note number (0–127)	velocity (0–127)
2	program change	1–16	–	program number (0–127)
3	start	–	–	–
4	stop	–	–	–
5	continue	–	–	–
6	key pressure	1–16	note number (0–127)	velocity (0–127)
7	channel pressure	1–16	–	velocity (0–127)
8	pitch bend	1–16	–	pitch bend (0–16383)

Figure 3: TouchOSC MIDI mappings.

voice messages having two data bytes, and the single data byte of channel pressure (aftertouch) and program change messages. The pitch bend message gets special treatment; in this case the value of the control variable is mapped to the entire 14 bit range of 0–16383. (In MIDI this value is the combination of the two data bytes of the message, hence the 14 bit value range.)

Considering a variable with the source (OSC) range x_1–x_2 and the target (MIDI) range y_1–y_2, the variable (OSC) value x is mapped to:

$$y = y_1 + (y_2 - y_1)\frac{x - x_1}{x_2 - x_1}.$$

In the case of the default OSC value range ($x_1 = 0$, $x_2 = 1$), this can be simplified to:

$$y = y_1 + (y_2 - y_1)x.$$

The resulting value y is then rounded to an integer and clamped to the MIDI data byte range (or the 14 bit range for pitch bend messages). By default, $y_1 = 0$ and $y_2 = 127$ ($y_2 = 16383$ for a pitch bend message).

For voice messages the user may configure the (MIDI) value range for the control variable, the (fixed) value of the MIDI channel and the (fixed) value of the *first* data byte of the message, if any.

Figure 3 summarizes the MIDI conversions supported in the latest TouchOSC version. The MIDI message type numbers in the first column are as given inside the XML layout file. (TouchOSC uses its own encoding for the message types which has nothing to do with the actual MIDI status bytes of these messages.)

Note that while it's possible to map a variable to the velocity of a note or the value of a control change message, you cannot map it to the note number or MIDI controller number. While this kind of setup might occasionally be useful, TouchOSC doesn't allow it. Still it's possible to implement most kinds of controller configurations, such as mixer interfaces, DJ controls and even MIDI keyboards without much trouble.

The (OSC) source value for a MIDI control can be any of the x, y, z and c status variables. In a multi-control widget, each of the subcontrols has its own MIDI assignments. The TouchOSC editor allows you to pick those from a dropdown list in the MIDI properties (in the left side pane of the editor) after selecting a widget.

Note that it's possible to map the color (c) variable as well, so that you can change the color of a widget by sending a corresponding MIDI message. In this case the MIDI value range is fixed at 0–8, where 0 denotes `red`, 1 `green`, etc. Another special case that deserves mentioning are mappings of the page messages (`/1`, `/2`, etc. in OSC). You can map any of the MIDI voice messages to a given page, so that a MIDI message will be emitted if the user switches tabs on the device, and the current page will be switched when the MIDI message is sent to the device.

Our implementation fully supports all types of MIDI assignments described above. Note, however, that in order to receive touch messages (z variable), the corresponding OSC message type must be enabled in TouchOSC's OSC configuration dialog.

4 Interfacing TouchOSC and Pd

Our TouchOSC MIDI Bridge does its job by converting OSC messages from/to MIDI and thus a working OSC connection between Pd and the device running TouchOSC is required. While Pd doesn't offer any built-in OSC support, this can easily be added by means of corresponding Pd externals (plugins). Two well-known external libraries for this purpose are OSCx and mrpeach. These are both included in

Pd distribution packages such as Pd-Extended[10] and Pd-L2Ork[11]. The mrpeach externals offer additional features, such as the ability to access the source address of an incoming OSC message which is useful in order to set up bidirectional communication in an automatic fashion. Our sample patches employ this feature and are thus written using the mrpeach externals.

To make these facilities available, you only need to make sure that you have the mrpeach externals installed and on your Pd library search path. This should already be the case if you're running Pd-Extended or Pd-L2Ork. The only other required setup is to verify your TouchOSC configuration on the device. In the OSC setup, the host address should be set to the IP address of the computer running Pd (under Linux, you can find this by running the `ifconfig` program). You should also verify that the outgoing and incoming ports in the TouchOSC configuration are set to 8000 and 9000, respectively, since these are the default values our sample patches assume. These port numbers match the TouchOSC defaults, however, so chances are that you only need to enter the correct host address on the device.[12]

5 The MIDI Bridge

Our TouchOSC MIDI bridge is distributed in the form of a Pd external library called `touchosc` which implements two objects `tomidi` and `toosc`. Both objects take the name of a TouchOSC layout as their single creation argument. Thus, for instance, to have OSC messages converted to MIDI using the MIDI assignments in a layout named `sample.touchosc`, you'd create the object in Pd as `tomidi sample`. To make this work, the layout file needs to be in the same directory as the Pd patch containing the object. You can also use a full path name including the `.touchosc` extension, enclosed in double quotes, as in `tomidi "/some/path/sample.touchosc"`.

The operation of the Pd objects is fairly straightforward and doesn't require any additional configuration. Both objects provide a single inlet and a single outlet. The `tomidi` object takes OSC messages on its inlet and produces the corresponding MIDI messages on its outlet. The `toosc` object does the reverse operation, mapping MIDI messages to their OSC counterparts. The conversion is fully automatic once you've configured the MIDI mappings in your TouchOSC layout. No manual processing of OSC messages is required.

Message Type	Format
control change	`ctl` v n c
note	`note` n v c
program change	`pgm` n c
key pressure	`polytouch` v n c
channel pressure	`touch` v c
pitch bend	`bend` v c
start	`start`
stop	`stop`
continue	`cont`

Figure 4: MIDI representation of the Pd TouchOSC bridge. n denotes the note or controller number, v the velocity or controller value, c the MIDI channel number.

OSC messages are represented in the same symbolic format that's also used by the OSCx and mrpeach externals, so the output of `unpackOSC` (mrpeach) or `dumpOSC` (OSCx) can be piped directly into `tomidi`, while the output of `toosc` is ready to be used with mrpeach's `packOSC` or OSCx's `sendOSC`.

MIDI messages are also encoded in a symbolic format, i.e., as Pd meta messages. As Pd doesn't have a standard representation of MIDI messages other than as a numbers graveyard, we invented our own, but it's fairly straightforward if you're familiar with Pd's objects for MIDI input and output. A summary of the message syntax can be found in Figure 4. The format and, in particular, the somewhat idiosyncratic order of arguments has been designed so that it's easy to dispatch on the different message types using a Pd `route` object and pass the remaining data to the corresponding MIDI output objects. Conversely, MIDI messages can be received from Pd's MIDI input objects and converted to our format by just packing together the data and tacking on the proper message selector. Two helper patches `midi-input` and `midi-output` are included in the distribution to do this.

The distribution also includes a helper patch named `touchosc-bridge` (cf. Figure 5) which takes care of all the nitty-gritty details of set-

[10] `http://puredata.info/downloads/pd-extended`

[11] `http://puredata.info/downloads/Pd-L2Ork`

[12] TouchOSC also supports Zeroconf, which is implemented in the latest versions of our software as well. This makes it much easier to set up the network connections, see Section 6. But if necessary you can also configure the network connection by manually entering IP addresses and port numbers as explained above.

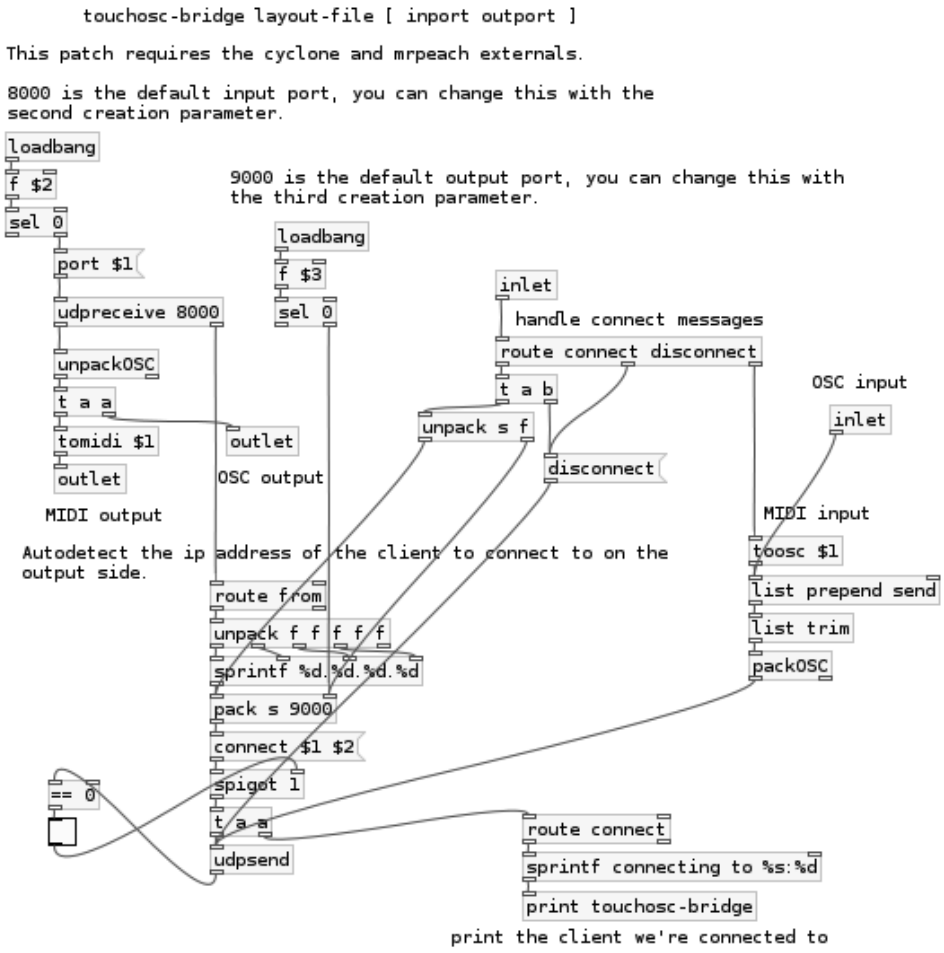


Figure 5: `touchosc-bridge` patch (simplified version).

ting up incoming and outgoing OSC connections, and provides a pair of `tomidi` and `toosc` objects to handle conversions in both directions. We describe this in the following section, where we cover the installation and usage of the MIDI bridge with Pd.

The latest versions of the `touchosc` library also include a third `oscbrowser` object which lets you discover available OSC clients, and can also publish its own OSC service using Zeroconf. This object is used to implement the Zeroconf support in the `touchosc-bridge` patch, but will also be useful in its own right for Pd users who wish to implement OSC applications; please check the pd-touchosc sources [3] for details.

6 Installation and Usage

Our TouchOSC MIDI bridge library for Pd is written in the author's Pure programming language [1], so you first need to install the Pure interpreter along with the pd-pure, pure-stldict and pure-xml addon modules. The Pure website will tell you how to do this. Binary packages for various popular Linux distributions such as Arch, Fedora and Ubuntu are also available, as well as ports for Mac OS X and BSD systems. Note that the pd-pure module is required to run any Pure externals with Pd, and needs to be enabled in Pd; please check the pd-pure documentation for details [2].

Next, to install our TouchOSC MIDI bridge, go find the pd-touchosc repository on Bitbucket [3] and clone the repository, or download it as a zip archive and extract it on your hard disk. At the repository website you can also find detailed installation instructions in the `README.md` file. Basically, you'll have to chdir to the source directory and run `make && sudo make install`. If you have Pd and all the other requisite software installed, this should build the external library and install it under your `/usr/lib/pd/extra` directory, along with some helper patches and exam-

ples.[13] Then fire up Pd and add `touchosc` to your startup libraries. Next time you start up Pd you should see a message in the Pd console showing that the `touchosc` library was loaded and registered with pd-pure.

Last but not least, you'll need TouchOSC, of course. You can grab the mobile app on Google Play or the iTunes Store and install it on your Android or iOS device. The TouchOSC editor can be downloaded for free on the TouchOSC website; it's a Java program, so you need to have a suitable Java runtime installed to use it.

The recommended way to run pd-touchosc is via the included `touchosc-bridge` helper patch. This patch sets up a pair of `tomidi` and `toosc` objects along with all the required OSC input and output machinery to connect with TouchOSC. The current version of the `touchosc-bridge` patch is depicted in Figure 5.[14] The patch is normally invoked with a single creation argument, the TouchOSC layout to use (in the same format as described in the previous section). Optionally, you can also configure the TouchOSC UDP ports by specifying these as the second and third argument.

The patch also detects the source IP address of incoming OSC messages and connects its output to it, so that after sending some data from the device the reverse connection should also work in an automatic fashion. Note that TouchOSC has an option which makes it send out OSC `/ping` messages in regular time intervals. If you enable this option then the `touchosc-bridge` patch will automatically set up its output connection as soon as it receives the first `/ping` message from the device.

Getting the network connections set up is even easier with Zeroconf. The latest version of pd-touchosc supports this via the Avahi Zeroconf daemon available for Linux and other Unix-like systems.[15] If you have Avahi installed and its daemon running, a client named `pd-touchosc` should show up in the host list in TouchOSC's OSC configuration dialog. Click on that to have the network address and port number filled in. On the host side, the full version of the `touchosc-bridge` patch offers the option to browse for available OSC services using Zeroconf. There's a toggle which lets you enable this; `touchosc-bridge` will then connect to the first OSC service available in the network (other than `pd-touchosc` itself). If there's more than one such service, you can cycle through the available services with the other GUI controls of the patch; see Figure 6. E.g., if you're running the Android version of TouchOSC then you'll have to look out for services named `Android (TouchOSC)` or similar (depending on which `ZeroConf Name` you chose in TouchOSC's OSC configuration) and pick the one that you want.

The left inlet/outlet pair of the patch is for sending MIDI messages to and receiving them from the device. In addition, the right inlet/outlet pair can be used to send and receive untranslated OSC messages. If you connect the left inlet and outlet to the `midi-input` and `midi-output` patches provided in the distribution, and route Pd's MIDI inputs and outputs to your MIDI devices and/or applications as needed, you should be able to set up Pd as a simple TouchOSC MIDI bridge with little effort. Or, if your computer music application is implemented as a Pd patch, you can use `touchosc-bridge` directly in your patch and hook it up to your existing control logic.

Figure 6 shows how `touchosc-bridge` can be employed in a simple test patch. Several sample patches and corresponding TouchOSC layouts are also included in the distribution. To get started, just download the sample layouts to your device, open the corresponding patches with Pd and kick the tires to see how things work.

7 Conclusion

We presented a TouchOSC MIDI bridge implementation which works on Linux, running inside Miller Puckette's Pd environment. This software allows you to convert between TouchOSC-formatted OSC and MIDI messages, following the MIDI mappings defined in a TouchOSC layout. It offers pretty much the same functionality as the official TouchOSC MIDI Bridge application, which is only supported on Mac and Windows at this time. The main differences to hexler's bridge are that it requires an actual OSC connection to the device and the TouchOSC layout file on the host side to work.

[13] This will work with vanilla Pd. For Pd-Extended and Pd-L2Ork you'll have to specify the Pd flavor using the `PD` make variable, e.g.: `make PD=pd-extended && sudo make install PD=pd-extended`.

[14] For the sake of clarity, the figure actually shows a simplified version of the patch, available in the distribution as `touchosc-bridge-simple.pd`, which doesn't include Zeroconf support. The full version of the patch can be found in the distribution as `touchosc-bridge.pd`.

[15] `http://avahi.org/`

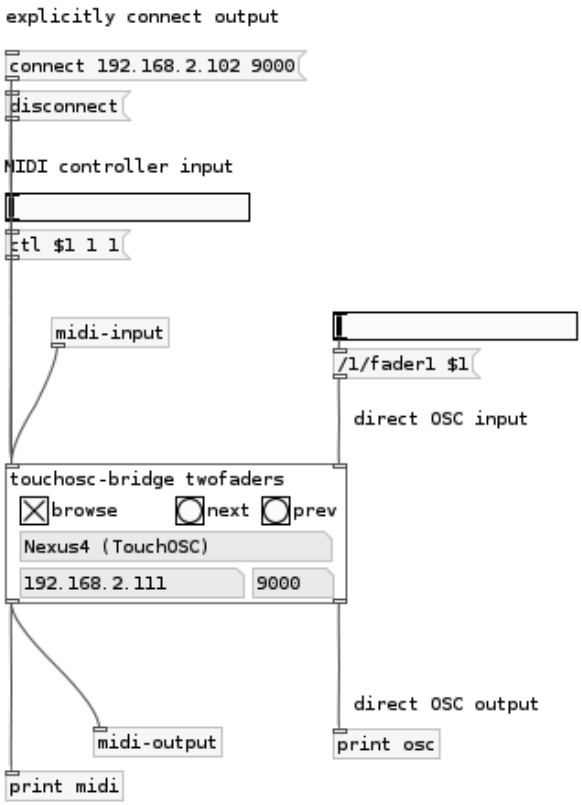


Figure 6: Sample patch using the `touchosc-bridge` abstraction.

To the author's knowledge, this is the first (and at the time of this writing, the only) fully automatic TouchOSC MIDI bridge application on Linux. Compared to the "real" TouchOSC MIDI Bridge, it also has the advantage that it is available under an open source license and doesn't rely on any proprietary and undocumented protocols, so it can easily be customized by the user.

Future work should be directed towards turning the software into a stand-alone program which can be run more easily by non-Pd users. In principle, it should also be possible to adjust our implementation to other OSC applications such as Control and the Lemur, although this will require some refactoring of the layout parsing code.

References

[1] A. Gräf. Signal processing in the Pure programming language. In *Proceedings of the 7th International Linux Audio Conference*, Parma, 2009. Casa della Musica.

[2] A. Gräf. pd-pure: Pd loader for Pure scripts. `http://puredocs.bitbucket.org/pd-pure.html`, 2014.

[3] A. Gräf. pd-touchosc: TouchOSC MIDI Bridge for Pd. `https://bitbucket.org/agraef/pd-touchosc`, 2014.

[4] hexler. TouchOSC: Modular OSC and MIDI control surface. `http://hexler.net/software/touchosc`, 2014.

[5] JazzMutant. Multitouch controllers for audio production, live music and media performance. `http://www.jazzmutant.com`, 2014.

LV2 Atoms: A Data Model for Real-Time Audio Plugins

David E. Robillard
School of Computer Science, Carleton University
1125 Colonel By Drive
Ottawa ON K1S 5B6
Canada
d@drobilla.net

Abstract

This paper introduces the LV2 *Atom* extension, a simple yet powerful data model designed for advanced control of audio plugins or other real-time applications. At the most basic level, an atom is a standard header followed by a sequence of bytes. A standard type model can be used for representing structured data which is meaningful across projects. This model is currently used by several projects for various applications including state persistence, time synchronisation, and network-transparent plugin control. Atoms are intended to form the basis of future standard protocols to increase the power of host:plugin, plugin:plugin, and UI:plugin interfaces.

Keywords

LV2, plugin, data, state, protocol

1 Introduction

LV2 is a portable plugin standard for audio systems, similar in scope to LADSPA, VST, AU, and others. It defines a C API for code and a format for data files which collectively describe a plugin. The *core* LV2 API is similar in power to LADSPA, but *extensions* can support more advanced functionality. This allows the interface to evolve and accommodate the needs of real software as they arise.

LV2 is supported by many applications, including Digital Audio Workstations like Ardour [Davis and others, 2014], hardware effects processors like MOD [Ceccolini and Germani, 2013], and signal processing languages like Faust [Gräf, 2013].

One key piece of functionality LV2 adds to LADSPA is the ability to transmit events. This is most commonly used to communicate via MIDI [MID, 1983] for playing notes, selecting programs, etc. MIDI is nearly ubiquitous in musical equipment, but has significant limitations [Moore, 1988]. Many applications require a more powerful model to express and manipulate state. For example, "load sample `/media/bonk.wav`", a typical operation in some audio software, can not be expressed in standard MIDI. Other protocols like OSC [Wright, 1997] are more powerful, but still designed around *commands*, which limits their applicability and ability to express structured data.

This paper introduces the LV2 *Atom* extension [Robillard, 2012b], a simple yet powerful data model designed for advanced control of LV2 plugins or other real-time applications. Atoms serve both as a model for representing state, and a protocol for accessing or manipulating it. This includes primitive values like numeric controls or file names, but the model-based approach allows developers to work with more sophisticated data as well.

The key distinction between MIDI or OSC messages and atoms is that atoms are not just commands, but a general data format. This paper aims to show that building on the foundation of a solid data model is more elegant and powerful than command-based protocols. The idea is conceptually similar to the popular use of JSON [Crawford, 2006] in the web community: define a data model for representing arbitrary information, then construct messages *within* that data model.

However, atoms are not introduced to the exclusion of other protocols. In fact, MIDI messages are transmitted to and from plugins as a particular type of atom. At the lowest level, atoms are a sequence of bytes (or a *chunk*) with a standard header. On top of this, a type model is defined which allows complex structures to be built from a few standard primitive and container types. This model has several advantages, including extensibility, support for round-trip portable serialisation, and natural expression in plugin data files.

There are two aspects to the LV2 atom specification: the low-level mechanics (Section 2) define the binary format of atoms and how they may be used, while the high-level semantics (Section 3) define a type model built upon this binary format. Using this model, projects can communicate meaningful structures at a conceptually high level, while the actual mechanics involved are simply the copying of small chunks. This approach to plugin control has many applications (Section 4) in current projects, which typically use the provided convenience APIs

for reading and writing atoms (Section 5) with ease. Ultimately, atoms are intended to form the basis of future work (Section 6) designing standard protocols for advanced plugin control.

2 Mechanics

2.1 Atom Definition

An LV2 atom is a 64-bit aligned chunk of memory that begins with a 32-bit size and type:

```
typedef struct {
        uint32_t size;
        uint32_t type;
} LV2_Atom;
```

This *atom header* is immediately followed by the *body* which is `size` bytes long. Atoms are, by definition, Plain Old Data (POD): contiguous chunks of memory that can safely be copied bytewise.[1] At the most basic level, this is all there is to atoms.

Types are assigned dynamically and not restricted to any fixed set. Developers can define new atom types, though all types are required to be POD. Any atom can thus be copied or stored, even by an implementation which does not understand its type. Among other advantages, this makes it possible for hosts to transmit atoms between plugins without explicitly supporting each type used. Similarly, generic plugins like event routers, multiplexers, or delays, can work with any atom. Developers are free to send complex data between plugins, or between UIs and plugins, without being held back by lacking standards or host support. Section 3.3 explains in detail how this decentralised extensibility is achieved.

Note, however, that atoms are only POD by definition, not necessarily portable: atoms may contain architecture-specific data like integers with native endianness. The atom specification includes a set of standard types which should be used where persistence or interoperability are important (see Section 3.1).

2.2 Communication via Ports

Plugins can send or receive atoms via an `AtomPort` which (like any LV2 port) is either an input or an output. An `AtomPort` is connected directly to an `LV2_Atom` (just as a standard LADSPA or LV2 control port is connected directly to a `float`).

An `AtomPort` can be used with any atom type. Plugins can specify which types are supported using the `atom:bufferType` property in their data files. Several types may be supported by a single port.

Input logistics are straightforward: the host connects the input to an atom before calling the plugin's `process()` method.

Outputs are slightly trickier since the plugin must know how much space is available for writing, but atom types may have variable size. To resolve this, the host initialises the `size` field in the output buffer to the amount of available space before calling `process()`. Plugins read this value, then write a complete atom (including `size` and `type`) to the buffer before returning. For real-time support, as with audio, output buffer space is made available by the host before calling `process()`. By default, outputs are given the same amount of space as inputs of the same type, but plugins that require more space can request larger output buffers ahead of time.

Thus far, atoms have been described without referring to specific types. An `AtomPort` can be connected to any value, but since plugins process signals over a block of time, it is usually more useful for ports to contain many time-stamped atoms, or *events*. To achieve this, ports are connected to a `Sequence` atom. This is the mechanism commonly used by LV2 plugins to process streams of sample-accurate events (including MIDI) alongside audio. The following section describes the set of standard atom types, which includes primitives like `Int` and containers like `Sequence`.

3 Semantics

3.1 Atom Types

The structure of the atom type model is similar to JSON values or ERLANG terms [Virding et al., 1996]: a few primitive types, and collections which can be used to build larger structures. The hierarchy of standard types defined in the atom extension[2] is shown in Figure 1.

Primitives represent a single value, and do not contain other atoms. The simplest types are primitives with fixed size, like `Int`. These types have a corresponding C struct defined in `atom.h` which completely describes their binary format, for example:

```
typedef struct {
    LV2_Atom atom;
    int32_t  body;
} LV2_Atom_Int;
```

The other `Number` types and `Bool` correspond to the C types with the same name, but have a precisely defined size on all platforms (32 and 64 bits).

[1] Type 0 has been reserved for a special reference type, in case a need for non-POD communication arises in the future.

[2] There is also a standard type for MIDI messages defined in the separate LV2 MIDI extension.

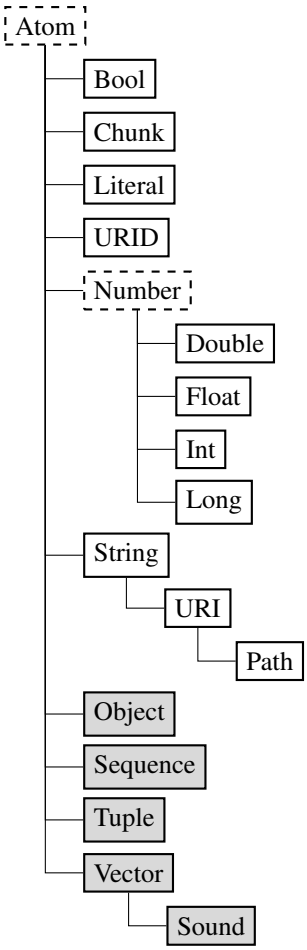


Figure 1: The Atom type hierarchy. Abstract types are dashed, and collections are grey.

A `URID` is a URI which has been mapped to a 32-bit integer by the host. This facility allows URIs to be used conceptually, but with the performance of fixed-size integers (Section 3.3 explains the purpose of URIDs in more detail).

Other primitive types have variable size. `String` and `Literal` represent raw strings and string literals with a datatype or language, respectively. A `Chunk` contains an opaque chunk of data.

Larger structures can be built from these primitives using collections. The most basic collection type is `Tuple`, a heterogeneous series of atoms of any type. An `Object` is a set of *properties*, each with a URID *key* and a *value* of any type. `Tuple` and `Object` are analogous to arrays and objects in JSON, or tuples and dictionaries in Python [van Rossum, 2010], respectively: universal containers that can express almost any structured data.

The remaining collection types are essentially optimisations for audio applications. A `Sequence` is, like `Tuple`, a series of atoms, but each is preceded with a time stamp.[3] A `Vector` is a series of fixed-length atoms with the same type and no headers, making the vector body a regular C array. `Sound` is a descriptive type, identical in format to a `Vector` of `float`, but explicitly representing a sample of audio.

3.2 Portable Serialisation

In addition to a binary format, each atom type has a portable serialisation. This allows implementations (typically hosts) to convert atoms to and from text for portable storage, network transmission, or human readability. This format is used to describe atoms in the following sections, but it is important to keep in mind that plugins work with atoms in their native binary form.

Most primitive types are associated with XSD [W3C, 2004b] datatypes which define their textual format. Table 1 shows this mapping along with an example string. URID is omitted since the portable serialisation of a URID is a URI.

Atom	XSD	Example
Bool	boolean	true
Chunk	base64Binary	vu/erQ==
Double	double	2.99e8
Float	float	0.6180
Int	int	-42
Long	long	4294967296
String	string	hello
URI	anyURI	http://lv2plug.in/
Path	anyURI	/home/drobilla/

Table 1: Text serialisation for primitives.

Containers and `Literal` have an abstract RDF [W3C, 2004a] serialisation which can technically be written in many formats. Here, the syntax of choice is Turtle [Beckett and Berners-Lee, 2011], which is used in LV2 data files.

All containers have a portable serialisation, but this paper focuses on the use of `Object`. The format for the other containers is omitted for brevity, but can be found in the LV2 atom specification.

An object in Turtle begins with its ID, followed by properties separated with semicolons. A "." terminates the description. For example, an `Object` named `eg:control` with three properties can be written as:

```
eg:control
    lv2:minimum 0.0 ;
    lv2:maximum 1.0 ;
    lv2:default 0.5 .
```

[3] Currently time stamps are always in samples, though other units are possible.

The ID and properties shown here are abbreviated URIs, for example, `lv2:minimum` is actually the URI `http://lv2plug.in/ns/lv2core#minimum`. A full Turtle document has prefix directives to define these precisely.

Numbers are shown unquoted, which is valid but does not precisely map to Atom types (e.g. 1.0 could be a `Float` or a `Double`). To preserve type in a machine serialisation, explicitly typed literals like `"1.0"^^xsd:float` are used instead.

3.2.1 Serialisation in Practice

A text-based format for describing atoms facilitates discussion, but is also useful in practice. The Sratom [Robillard, 2012c] library provides a simple C API for lossless round-trip serialisation of any atom built from the standard types. This is used in several different scenarios:

- Saving plugin state in sessions, which is supported by many hosts.
- Jalv [Robillard, 2012a], a single-plugin host for Jack [Davis, 2001], can log all communication between plugin and UI to the console. This is particularly useful for debugging.
- Ingen [Robillard, 2014], a modular plugin host and plugin itself, has a UI that communicates to the engine exclusively via atoms. When running as a plugin, binary atoms are sent via `AtomPort`, but the UI can also run remotely by communicating over a TCP socket in Turtle. This way, UIs on different architectures can control the engine, including those written in non-C languages like Python or Javascript.

3.3 URIs and Extensibility

Types and properties are identified by URI. The benefit of URIs is that anyone can define new terms without needing to worry about clashes or centralised coordination.

In the context of LV2 atoms, this allows developers to invent new types and properties without requiring "approval". This freedom is particularly useful while developing new ideas, be they experimental, for internal use only, or intended for eventual standardisation.

For example, the previous sections use the `lv2:minimum` property, but suppose a plugin developer additionally needs to describe a "sweet spot" for controls. There is no standard LV2 property for this concept, so the developer can define their own (e.g. `http://drobilla.net/ns/sweetSpot`), use it in their data files, implement host support if necessary, send it between plugins or between plugin and UI, and so on. The implementation can be tested and released to the public without any binary compatibility issues. This flexibility allows LV2 to evolve to meet real developer needs with minimal friction.

Note that URIs here are simply serving as global identifiers, and are not required to actually resolve on the Internet. However, developers should use URIs in domains where they *could* host pages, since this avoids potential conflicts.[4] There is no need to own an entire domain, for example many plugins use URIs at popular project hosting sites.

URI schemes other than HTTP may be used, but are not recommended. One advantage of HTTP is the ability to have URIs resolve to useful resources, particularly documentation. All standard LV2 URIs work this way, so documentation is often just a click away (follow the above `lv2:minimum` URI for an example). The LV2 distribution includes a tool, `lv2specgen`, which generates documentation for types and properties which are defined in Turtle.

4 Applications

4.1 Time

The most common use of objects to communicate between plugins and hosts is transport synchronisation. To keep plugins updated with tempo information, hosts send an object with properties describing the current time and tempo, whenever changes occur.

Most hosts send updates that roughly correspond to Jack transport information, but with floating point beats instead of PPQN ticks, and a single floating point speed instead of only "rolling" or "stopped". For example:

```
[]
    a                   time:Position ;
    time:frame          88200 ;
    time:speed          0.0 ;
    time:bar            1 ;
    time:barBeat        0.0 ;
    time:beatUnit       4 ;
    time:beatsPerBar    4.0 ;
    time:beatsPerMinute 120.0 .
```

The "a" here is Turtle short-hand for "is a" or "type", equivalent to the `rdf:type` property.

4.2 UI Communication

Atoms are also useful for communicating with components other than the host. The most common of these in practice is communication between a plugin

[4] Inventing URIs under other domains without permission is inappropriate!

and a custom UI (which, in LV2, happens via ports). Many UIs need to perform more advanced operations than is possible via `float` control ports. For example, a plugin may include an envelope with an arbitrary number of points, which a UI could control with messages like

```
[]
    a         eg:EnvelopeSegment ;
    eg:endX   1.6 ;
    eg:endY   0.5 ;
    eg:shape  eg:linear .
```

Several projects have made use of such messages for controlling plugins from custom UIs. While host-transparent (and thus automatable) control is preferable, full control of some plugins requires messages that are not currently standardised (e.g. LV2 presently has no concept of envelope segments, or multi-dimensional controls in general). However, though the message does not have standardised semantics, it is built from standard atom types so that hosts can make some sense of it. In particular, hosts can serialise such messages for controlling a plugin running on a remote computer or embedded device. This is a good example of how an extensible model allows developers to achieve their goals without being held back by lagging standardisation or host support.

In the future, standardised message types will allow plugins and UIs to use event-based control with host support for friendly interfaces and automation, where appropriate.

4.3 Plugin State

LV2 has a *state* extension which allows plugins to save and restore state beyond control port values. The state extension does not directly depend on the atom extension, but has a property-based API that meshes naturally with `Object`. Plugins use host-provided callbacks to save/restore a URID key, `void*` value, and URID type.

The fact that plugin state and `Object` are both based on properties suggests an elegant approach to plugin design: one set of properties can serve both as plugin state and real-time control protocol. This means plugin developers do not need to design both a state model and protocol, but simply define a set of properties that describes their plugin's state.

For example, the sampler example plugin included with LV2 can play any `.wav` file, and the sample can be loaded by sending a message like:

```
[]
    a              patch:Set ;
    patch:property eg:sample ;
    patch:value    </media/bonk.wav> .
```

The `patch:Set` type and properties used here are defined in the LV2 *patch* extension, which defines several message types for getting and setting property values.

The `eg:sample` property is saved as part of the sampler's state. Thus, this single property is used to both control the plugin and represent a value in saved state. There is no need to define both a special "set sample" command *and* a format for saving that information.

4.4 Properties

Developers can invent new property URIs and use them in code without defining anything. However, it can be useful to define properties for documentation purposes, and in some cases host support.

Properties are defined in Turtle, so they can be included alongside plugin descriptions. For example:

```
eg:sweetSpot
    a             rdf:Property ;
    rdfs:domain   lv2:ControlPort ;
    rdfs:range    xsd:float ;
    rdfs:label    "sweet spot" ;
    rdfs:comment "The nicest value." .
```

Defining properties in this machine-readable format is mainly useful for generating documentation (all standard LV2 properties are defined in this way), but this information can be used by hosts as well.

This area is still experimental, but for example, Jalv will show a file selector in its host-generated UI for plugins that support properties with `Path` values. Setting the property is achieved by sending a `patch:Set` message like the example shown in the previous section.

4.5 Presets and Default State

Plugin descriptions can include a set of default state properties which should be loaded initially. A preset has a similar structure to a plugin description, and can also include state. This means that presets can not only set port values, but restore arbitrary internal plugin state like loaded samples. The benefit of using standard atom types to describe state is that developers can write default state in plugin data files, and hosts can serialise state/presets in the same format. For example, a preset for a sampler can specify a sample to load like so:

```
eg:clickeyPreset
    lv2:appliesTo eg:sampler ;
    # ...
    state:state [
        eg:sample <click.wav>
    ] .
```

5 Reading and Writing Atoms

It's convenient to think of atoms in high level terms, and describe objects in human-readable Turtle, but plugins are typically written in C and must work with binary atoms. For simple primitive types like `Int` this is trivial: the appropriate structs can be created, copied, and read in the usual way.

Collections are more complex, since their bodies have variable size and possibly an irregular and/or nested structure. To make reading collections easier, iterators for each collection type are provided in a utility header.

For objects, iteration works, but is tedious and verbose in the typical case of getting a few property values. To make this case more succinct, a simple accessor for object properties is provided. For example, consider an object that describes a 2D point:

```
[]
    a    eg:Point ;
    eg:x 1.0 ;
    eg:y 2.0 .
```

If `obj` points to this object, and `ids` contains the necessary mapped URIs, the `eg:x` and `eg:y` values can be accessed like so:

```
const LV2_Atom* x = NULL;
const LV2_Atom* y = NULL;
lv2_atom_object_get(obj,
                    ids.eg_x, &x,
                    ids.eg_y, &y,
                    0);
```

Here, `x` and `y` are pointed directly at the corresponding values within `obj`. There is no dynamic allocation, so this code is real-time safe and does not require the user to clean up `x` and `y`. If the object does not have a matching property, the result will be `NULL`. Note that this code will continue to work correctly even if additional properties are added to the object in the future.

Writing collections can be trickier, particularly those with nested structure. For example, an `Object` property may have a `Tuple` or another `Object` as a value. Atoms can be constructed in-place by repeatedly appending to a buffer, but correctly maintaining container `size` fields and padding requirements can be a delicate task. To make writing simple, a *forge* API is provided which allows arbitrarily complex atoms to be constructed in a target buffer. The forge has a method for each atom type: for primitives it simply appends the given value, and for containers it appends the atom header and returns a *frame* which must be popped when the object is finished. Container sizes are updated automatically as atoms are written using this stack of frames. The forge is safe to use in real-time code, and can be used by plugins to write objects directly to `AtomPort` outputs in their `process()` method. For example, the same 2D point object can be written like so:

```
// Begin an anonymous eg:Point object
LV2_Atom_Forge_Frame frame;
lv2_atom_forge_object(
    forge, &frame, 0, ids.eg_Point);

// eg:x 1.0
lv2_atom_forge_key(forge, ids.eg_x);
lv2_atom_forge_float(forge, 1.0);

// eg:y 2.0
lv2_atom_forge_key(forge, ids.eg_y);
lv2_atom_forge_float(forge, 2.0);

// Finish object
lv2_atom_forge_pop(forge, &frame);
```

6 Conclusions and Future Work

The LV2 Atom specification defines a simple binary format for any type of data, and an expressive type model for representing structured data within that format. This model has proven effective for representing plugin state, host to plugin communication such as tempo synchronisation, and custom control protocols such as between a plugin and its UI.

This work has laid the foundation for more powerful control of plugins and other real-time applications. There are two main areas of future work: additional convenience APIs and tools to make working with atoms as simple as possible, and building more advanced control protocols and other functionality using the atom model.

For convenience, the existing APIs described in Section 5 do a relatively good job of making it easy to construct and inspect atoms in C. However, some developers have found the forge confusing. It is difficult to make a fully capable writing API much simpler given the constraints of C and hard real-time, but one idea is to make a writing counterpart to `lv2_atom_object_get()` which works only for

non-nested objects. Using C++, a similar, but more elegant and type-safe interface would be possible, which could work even for nested containers. LV2 is defined in C, but a significant portion of the developer community uses C++, so a C++ convenience wrapper (including idiomatic iterators) would be a welcome improvement. Other minor improvements could ease the mechanics, but since several developers have successfully made use of atoms, focusing on this area may not be an effective use of time.

The other, more interesting, area for future work is building on the foundation of atoms to create more powerful control protocols. One of the biggest limitations of LV2 is the `ControlPort` inherited from LADSPA. Control ports can only hold a single float value, and tie the control rate to how often `process()` is called. This can be problematic for certain types of plugins. The lack of a mechanism for adding and removing ports also means that the set of controls is fixed, which prevents many possibilities such as the multi-point envelope example in Section 4.2. Using events for control instead of control ports can solve all of these problems. Events are much more powerful than a low-rate control signal, and allow a sample-accurate stream of changes to be sent to a plugin for an entire `process()` call. Logistically, this can be achieved via the current `AtomPort` + `Sequence` mechanism, but the structure of events required is yet to be determined. `Object` will likely form the basis for future standard messages, due to its inherent meaningfulness and extensibility. There are many possibilities opened up by moving to events, including ramped/smoothed controls, gestures, precise voice control, and note-specific modulation/articulation. This is one of the most exciting frontiers of LV2 development; a powerful event-based control scheme will enable new functionality beyond the current capabilities of host-agnostic plugins.

References

David Beckett and Tim Berners-Lee. 2011. Turtle - Terse RDF Triple Language. http://www.w3.org/TeamSubmission/turtle/. W3C Team Submission.

Gianfranco Ceccolini and Leonardo Germani. 2013. MOD - an LV2 host and processor at your feet. In *Linux Audio Conference 2013 Proceedings*, LAC 2013, pages 157–161. Institute of Electronic Music and Acoustics (IEM).

Douglas Crawford. 2006. The application/json media type for JavaScript Object Notation (JSON). http://www.ietf.org/rfc/rfc4627. RFC 4627.

Paul Davis et al. 2014. Ardour Digital Audio Workstation. http://ardour.org/.

Paul Davis. 2001. JACK Audio Connection Kit. http://jackaudio.org/.

Albert Gräf. 2013. Creating LV2 plugins with Faust. In *Linux Audio Conference 2013 Proceedings*, LAC 2013, pages 145–152. Institute of Electronic Music and Acoustics (IEM).

1983. *Musical Instrument Digital Interface Specification 1.0*. International MIDI Association. http://www.midi.org/techspecs/.

F Richard Moore. 1988. The dysfunctions of MIDI. *Computer Music Journal*, 12(1):19–28.

David E. Robillard. 2012a. Jalv. http://drobilla.net/software/jalv/.

David E. Robillard. 2012b. LV2 Atom. http://lv2plug.in/ns/ext/atom/.

David E. Robillard. 2012c. Sratom. http://drobilla.net/software/sratom/.

David E. Robillard. 2014. Ingen.

Guido van Rossum. 2010. *The Python Language Reference*. Python Software Foundation. http://docs.python.org/reference/.

Robert Virding, Claes Wikström, and Mike Williams. 1996. *Concurrent Programming in ERLANG*. Prentice Hall, second edition. http://www.erlang.org/.

W3C. 2004a. Resource Description Framework (RDF): Concepts and Abstract Syntax. http://www.w3.org/TR/rdf-concepts/. W3C Recommendation.

W3C. 2004b. XML Schema Part 2: Datatypes. http://www.w3.org/TR/xmlschema-2/. W3C Recommendation.

Matthew Wright. 1997. Open Sound Control - a new protocol for communicationg with sound synthesizers. In *Proceedings of the 1997 International Computer Music Conference*, pages 101–104.

Muditulib, a multi-dimensional tuning library

Funs SEELEN
http://www.muditulib.eu
Utrecht,
European Union,
mail@funsseelen.eu

Abstract

The *Muditulib* library is introduced and explained. Muditulib is mainly a library consisting of a collection of C header files that include functions written for the purpose of tuning tonal music within the diatonic scale. This scale, as well as the library's functions, along with pitch representation systems, will be explained in detail or just shortly with reference to other literature. A music theoretical background is useful, though not necessary. Along with the muditulib core functions an implementation for Pure Data is published. Developers are encouraged to write implementations for other synthesizers, music production platforms or any other link in the chain of tonal music production workflow.

Keywords

Tuning systems, pitch representation / MIDI, software library.

1 Introduction

Muditulib is developed to make the tuning of the common western (diatonic) scale easier, without being restricted to equal temperament of twelve tones per octave with a frequency ratio of $2 : 1$[1]. Music theorists and mathematicians have developed many tunings for this scale through the ages. Modern software like for example SCALA[2] can map all possible tunings to MIDI notes. In a flexible environment like Pure Data[3] one could rather easily implement such mapping oneself, so that is not the purpose of the library. Muditulib doesn't really map fixed scales to MIDI notes, but offers multiple methods to tune notes or intervals more dynamically. In that respect Muditulib is more familiar to the Hermode tuning system[4], although the approach is quite different. Both SCALA and Hermode will not be further explained here, for that is beyond the purpose of this paper. The next sentence deserves its own emphasized paragraph.

Muditulib has got nothing to do with microtonality, microtonal music, or microtones, whatever may be meant by those obfuscating terms.

This document is rather intended as an explanation of the software library Muditulib, along with a short summary of my research within the field of tuning and music theory, than purely as a genuine scientific article that describes research goals, methods, and conclusions. Its purpose is to propose several tuning and pitch representation systems to an audience of music software developers.

2 The diatonic scale

In order to understand the approach described here it will be helpful to explain a little bit about the diatonic system, particularly the distinction of variable steps in a scale. This is done most easily by freely citing a recent work by the present author in the next two paragraphs [Seelen, 2014].

The terms chromatic and diatonic descend from the old Greek musical system. Together with the enharmonic they formed the three tetrachords the Greek musical scales were made up from [Grout and Palisca, 1988, ch. 1]. The ancient tuning theory is clearly described by J. Murray Barbour [Barbour, 2004, ch. II]. Today's use of those terms is somehow related to that of their namegivers, although the tetrachord itself lost its value. The diatonic scale is a scale that consists of seven intervals or steps. The eighth note, or the octave, is a repetition

[1] The standard equation for translating MIDI notes to frequency is $f = 440 \cdot 2^{((m-69)/12)}$, where m is the MIDI note number and f is the frequency in cycles per second. In this tuning a diatonic semitone equals a chromatic semitone. Moreover, expressed as frequency ratios, a semitone equals the square root of a whole tone, thus, on a logarithmic scale, the semitone equals half a whole tone.

[2] http://www.huygens-fokker.org/scala/

[3] http://puredata.info/

[4] http://www.hermode.com/index_en.html

of the first one, usually with a frequency ratio of 2 : 1. Those seven steps are divided into five larger ones, the whole tones, and two smaller ones, the semitones. The scale then created is actually the same as two Greek diatonic tetrachords on top of each other, at one side overlapping (*conjunct*) and at the other side separated by one whole tone (*disjunct*). By the chromatic scale, however, usually a division of the octave in twelve equal parts is meant, which is quite different from an accumulation of Greek chromatic tetrachords.

So far I described the historical outlines of the system. In the frequency domain the relation between the octave x and whole tone T and semitone s is as shown in equation 1, where $1 < s < T$.

$$x = T^5 \cdot s^2 \tag{1}$$

The citation [Seelen, 2014] ends here. For further reading I refer to the mentioned article. The main point is that the tonal system used as a starting point for the tuning system is a 7-tone and not a 12-tone system, as western tonal music is often incorrectly described as. This idea of a 12-tone system just evolved from practical tuning matters concerning the 7-note system. For clarity: the diatonic scale is a theoretic scale rather than a scale of fixed frequency relationships[5].

3 Ts, a two-dimensional pitch representation system

Traditional western music notation is based on seven syllables[6] or alphabetical characters[7], which correspond to the graphical notes. In contrast to a one-dimensional representation like MIDI note numbers, western music notation makes a clear distinction between for example a C-sharp and a D-flat. All theoretic tone intervals consist of a number of whole tones and semitones, instead of just a number of *chromatic* semitones[8]. The system can therefore be interpreted as two-dimensional. In the previously mentioned article [Seelen, 2014] the $Ts()$ tonal representation system is proposed, consisting of the two values T_n and s_n, the number of whole tones and semitones, respectively. Its advantage is that it can be easily translated to the MIDI note system as is shown in equation 2. Its reference ($Ts(0,0)$) is set equal to the C corresponding to MIDI note 0. Therefore $Ts(25,10)$ corresponds to middle C (lilypond: *c'*).

$$m = 2 \cdot T_n + s_n \tag{2}$$

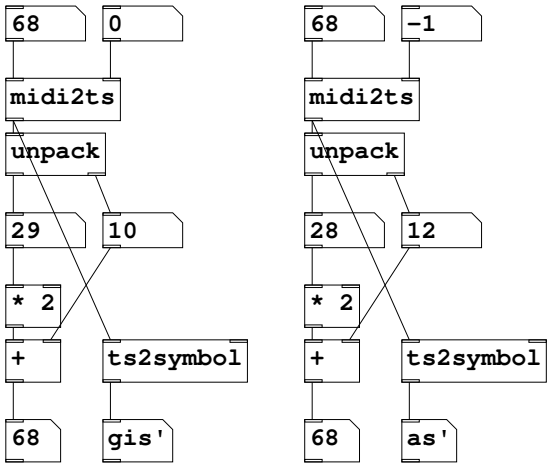


Figure 1: Examples of [midi2ts] and [ts2symbol] in Pure Data.

3.1 Ts to note name symbol

The translation of $Ts(T_n, s_n)$ to a note name symbol (the *Lilypond*[9] standard) is done by translating the total number of steps ($T_n + s_n$) to the root character plus octave designation[10]. Then the deviation from the reference of the root character is calculated and translated into an amount of flattening or sharpening.

3.2 Ts to frequency

This translation can be summarized to equation 3, where f is the frequency, A is the reference frequency for $Ts(29,11)$, corresponding to the note *a'*, x is the frequency ratio for the octave, and r is equal to $\frac{\log s}{\log T}$ and represents the semitone to whole tone ratio.

$$f = x^{((T_n-29)+(s_n-11)\cdot r)/(5+2\cdot r)} \cdot A \tag{3}$$

Usually x is set to 2. The ratio r then defines the kind of tuning. When r is set to about 0.6 the tuning could be said to be within the

[5] E.g. the diatonic perfect fifth can be tuned to $\frac{3}{2}$, as well as $2^{(7/12)}$, and many other frequency ratios.

[6] do-re-mi-fa-so-la-ti

[7] C-D-E-F-G-A-B

[8] The word *chromatic* is emphasized for the reason that this term can be interpreted in various ways and is therefore confusing. In this case one twelfth of an octave is meant.

[9] `http://lilypond.org/`

[10] Division by seven and its remainder (modulo).

mean tone zone. $r = 0.6$ or $r = \frac{3}{5}$ corresponds to 31-TET, for each semitone is made up from 3 and each whole tone from 5 dieses[11]. The exact meantone ($T = \frac{1}{2}\sqrt{5}$) temperament, Pythagorean, and other examples and how their parameters are calculated are shown in table 1.

Tuning / Temperament	Equation / Calculation	Parameters
Mean tone	$\frac{5}{4} = 2^{2/(5+2r)}$	$r \approx 0.60628$
Pythagorean	$\frac{3}{2} = 2^{(3+r)/(5+2r)}$	$r \approx 0.44247$
	$r = \frac{\log s}{\log T} = \frac{\log \frac{256}{243}}{\log \frac{9}{8}}$	
Tritone temperament	$\frac{7}{5} = 2^{(3/(5+2r))}$	$r \approx 0.59006$
Stretched octave, perfect 5th and 3rds	$\frac{5}{4} = \left(\frac{3}{2}\right)^{2/(3+r)}$	$r \approx 0.63412$
	$x = \left(\frac{5}{4}\right)^{(5+2\cdot r)/2}$	$x \approx 2.01246$
19-TET	$s = 2^{\frac{2}{19}}$, $T = 2^{\frac{3}{19}}$	$r = \frac{2}{3}$
31-TET	$s = 2^{\frac{3}{31}}$, $T = 2^{\frac{5}{31}}$	$r = \frac{3}{5}$
53-TET	$s = 2^{\frac{4}{53}}$, $T = 2^{\frac{9}{53}}$	$r = \frac{4}{9}$

Table 1: Tuning examples of the two-dimensional system.

3.3 MIDI note numbers to Ts

Ideally the MIDI note system is skipped in all translations of tonal data. The translation from Ts to MIDI, from two to one dimension as shown in equation 2, leads to irreversible data loss. The same applies to MIDI files exported from Lilypond. However, even if composed diatonic music material wouldn't be translated into MIDI anymore, still improvisations on MIDI keyboards should be interpreted by the computer. The simplest and probably best way of doing this is to leave the decision to the performer.

[11]The *diesis* is the interval that remains to the octave after an accumulation of three perfectly tuned ($\frac{5}{4}$) major thirds (e.g. *B-sharp* to *C*). This typically mean tone interval remainder is approximately a 31^{th} of an octave. A.D. Fokker uses this term to indicate the smallest interval in 31-TET [Fokker and Pol, 1942]

3.3.1 User-defined

In Muditulib this is done by setting a modulation parameter (*mod*). The default ($mod = 0$) is - as a starting reference - set to two flats (E and B) and three sharps (F, C, and G), similar to the baroque standard. Every modulation up replaces one note in the circle of fifths by adding $(1, -2)$ to its assigned Ts value, starting at MIDI note 3 (*E-flat* to *D-sharp*). In the opposite direction it starts at MIDI note 8 (*G-sharp* to *A-flat*). In the current implementation each modulation change is calculated from a default array at 'mod 0'.

3.3.2 Real-time pitch spelling

Another way to enrich the poor MIDI note data is the algorithmic approach. If, during a performance, a listener is able to roughly extract information about key, mode or tonality, the computer should be able too, if programmed according to a realistic cognition model. The translation from MIDI note data to staff notation or note names is called *pitch spelling* and is generally not a real-time practice. Some researchers have developed algorithms through the last decades [Longuet-Higgins and Steedman, 1971; Temperley, 2004; Cambouropoulos, 2003; Meredith, 2003; Chew and Chen, 2005; Honingh, 2006]. A very related topic is *key-finding*. In the end such procedures are all about saying something about the function of and relation between tonal events. For in some styles of music it is not always clear in which tonal direction the music will develop, when no true sense of tonality in that certain moment is present, a perfect real-time solution is theoretically impossible. The duty of the algorithm, however, is not offering perfect sheet music, but offering input for a real-time controlled dynamic tuning. Errors are acceptable, at least in cases where the human perception is uncertain. Any uncertain choice of the human tonal perception corresponds to the same uncertainty of the algorithm, ideally. An algorithm, developed by the present writer, based on memory, predicting, counting, averaging, and interval comparing, will be included in the library.

4 Tts, adding a third variable

Just intonation is a tuning approach in which all tone intervals are based on integer relationships. Pythagorean tuning can be considered 'just'. It is based on a perfect fifth $(3 : 2)$ and a perfect octave $(2 : 1)$ ratio. All intervals are then made up from powers of prime

numbers two and three[12]. As the mean tone temperament showed, however, the perfect major third ratio is 5 : 4, but adding the number five to the tuning system introduces a problem. A major third cannot be divided into two equal whole tones within just intonation, for the mean tone is not an interval based on an integer relationship. Therefore, the major third is divided into a large and a small whole tone. This way, thirds, both major and minor, perfect fifths as well as octaves, and therefore all octave inversions of the mentioned intervals, can be tuned correctly[13].

4.1 Tts to frequency

Tuning the Tts values is not a great deal, for there seems to be only one perfect solution, in which all octaves, perfect fifths, and thirds are tuned the most ideal way. Frequency ratios then should be $T = \frac{9}{8}$, $t = \frac{10}{9}$, and $s = \frac{16}{15}$. The translation to frequency can be best summarized by equation 4, where A is the reference *a'* at $Tts(17, 12, 11)$[14].

$$f = \left(\frac{9}{8}\right)^{(T_n-17)} \cdot \left(\frac{10}{9}\right)^{(t_n-12)} \cdot \left(\frac{16}{15}\right)^{(s_n-11)} \cdot A \tag{4}$$

However, still some adjustments are conceivable. The 53-TET system for example, with $r = \frac{4}{9}$ and therefore approximately Pythagorean tuning, can be divided into three variable steps. Instead of a semitone of four *commas*[15] and a whole tone of nine, the semitones are enlarged to five in favor of two of five whole tones. Note that the result of this would not deviate very much from the in equation 4 proposed tuning.

4.1.1 Turkish modes

An idea for further improvement would be to combine both two- and three-dimensional interpretations of 53-TET, or the three-limit Pythagorean and the five-limit just intonation instead. This would result in even more variables or steps, namely those used in Turkish modes (*makamlar*): *bakiye* (4 commas), *küçük mücennep* (5 commas), *büyük mücennep* (8 commas), *tanini* (9 commas), and the augmented second *artık ikili* (12 commas)[16] [Signell, 1986 1977]. Muditulib could then be made very suitable for digitally synthesized reproduction of Turkish classical music or anything alike.

4.2 Creating useful Tts data

In contrast to Ts data, Tts data cannot be extracted from regular scores when it concerns diatonic music. Tts values shall then be obtained from Ts or even MIDI. This raises the problem of the *syntonic comma*, that is, the difference between the large and the small whole tone ($\frac{81}{80}$). This means that to fit the needs of a certain interval, another interval might be tuned too wide. In a dynamic tuning this comma can be replaced on-the-fly. In a more fixed tuning the comma will stick to its initial place and be rather present. For example, from this point of view Turkish modes are based on the placement of syntonic commas, giving each makam its very own character, based on some slightly wider and narrower intervals[17]. Only fifths and octaves are always tuned into perfection. One could possibly write a book about the placement of the syntonic comma. However, for this moment the present author prefers to skip such time-consuming research effort and focusses on two approaches, proposed in the next paragraphs. Again, there is a relatively simple approach and a more elaborate.

4.2.1 MIDI to Tts, user-defined

The simple approach is the user-defined key setting. The user defines the mode and the starting MIDI note. Currently two modes are available, one minor and one major, both displayed in table 2. Different modes can be created by moving the pattern to another reference MIDI note or, of course, by editing the source code or submitting a supported feature request[18]. For a piece

[12] This is called *three-limit*.

[13] According to renaissance counterpoint prescriptions all intervals considered consonant [Mann, 1987], whether perfect or imperfect, are now covered.

[14] That is, NOT the MIDI note number 69 is the reference.

[15] In contrast to 31-TET a part is called comma instead of diesis. This comma refers to the *syntonic* comma, which will be discussed later.

[16] The mentioned augmented second, that is, a perfect fourth (22 commas) minus two large semitones (5+5 commas), is clearly not unique here and can not be considered an extra variable. The only difference is that both Pythagorean (small) and five-limit (large) semitones appear. That makes a total of four different steps.

[17] An example of this is the *uşşak* makam, starting with the relatively small Pythagorean minor third (13 instead of 14 commas), although consisting of a small large whole tone (8) and a large semitone (5) from the tonic [Signell, 1986 1977].

[18] The pattern is best recognized by looking at the 'difference' column. The non-steps (places where no semitone or step occurs, e.g. 'T/s') are grouped just like the black keys on a keyboard.

in E minor one would usually choose 'MIDI note 4' as reference and '0' (minor) as mode. These default patterns are carefully chosen to enable the best standard modulations from the reference key, without resulting in too many 'misplaced' commas[19]. How these patterns were actually chosen is not discussed here, for the sake of not going into detail of music theoretical considerations too much. Furthermore, more research on this topic would be desirable, for example comparing these considerations to those of how frets on a saz are placed.

Minor (0)			Major (1)		
Dif.	Total	C.	Dif.	Total	C.
s	-	0	s	-	0
s	s	5	T/s	T/s	4
T/s	T	9	s	T	9
s	Ts	14	s	Ts	14
t/s	Tt	17	t/s	Tt	17
s	Tts	22	s	Tts	22
T/s	TTt	26	T/s	TTt	26
s	TTts	31	s	TTts	31
s	TTtss	36	s	TTtss	36
T/s	TTTts	40	t/s	TTtts	39
s	TTTtss	45	s	TTttss	44
t/s	TTTtts	48	T/s	TTTtts	48

Table 2: Tuning patterns for the MIDI keyboard: modes and modulation options from a reference tonic. Shown are the differences to the previous MIDI note and the total amount of distance to the reference in symbols and in commas.

4.2.2 A pattern matching approach, or the *hexahord analysis algorithm*

Another way of tuning is leaving this task to, again, a real-time controlling algorithm. For singers in the Middle Ages used Guido of Arezzo's hexachord to choose their pitches [Grout and Palisca, 1988], the hexachord seems very suitable for on-the-fly tuning purposes. The next question is how the hexachord should then be tuned. Hermann von Helmholtz has been very helpful to answer this question for he explains how medieval singers related each note of the hexachord to a reference and what differences exist between major (on *Ut*) and minor (on *Re*) modes [Helmholtz, 1896, ch. 18]. The resulting conclusions are displayed in table 3. The *Re* and *Sol* are placed one comma lower in minor mode. However, for modulation purposes the major hexachord is placed one comma lower than the minor altogether. The translation from Ts to Tts is done by pattern matching[20]. The choice between major and minor tuning of each individual hexachord is done by a tonic-finding algorithm.

	Ut	Re	Mi	Fa	Sol	La
Minor	0	8	17	22	30	39
	-	t	T	s	t	T
Major	-1	8	16	21	30	38
	-	T	t	s	T	t

Table 3: The tuning of the hexachord, displayed in commas.

5 Implementation

All the previously mentioned functionality will be bundled into one file, a collection of `C` functions[21]. These functions will be explained in a reference manual at `http://muditulib.eu`.

In essence this library is relatively small and simple. The challenging part is probably the implementation, depending on the environment. An implementation for Pure Data is ready yet and consists of `C` files written against the Pd-API to create a collection of separate Pd classes, along with the muditulib core functions, a *Makefile* based on the template by H.-C. Steiner, helpfiles and examples. An example of the Pd-implementation is shown in figure 1.

6 Concluding remarks

The tuning approaches described here highly depend on implementation possibilities. For a low-level music production environment like Pure Data there is actually no problem, although this requires quite some background knowledge from the user, both of music and tuning theory. Most popular electronic music production platforms, however, are mainly based on the MIDI note system. Tuning workarounds making use of 'pitch bend' are familiar to the present author, though not a satisfying solution. Plans are to develop a file format other than MIDI. More about such can be expected in the near future. Any suggestions about file

[19] E.g. the fourth of 23 commas on the subdominant in minor mode (from Tts to TTTtss).

[20] The pattern of the diatonic hexachord is T-T-s-T-T.

[21] `http://sourceforge.net/projects/muditulib/`

formats or implementations and especially questions arising from implementation ambitions, are very welcome.

7 Acknowledgments

My thanks go out to Albert Gräf, Gerard van Wolferen, Pieter Suurmond, and Hans Timmermans for supporting my research within this field, to Horus and its initiator Marc Groenewegen for contributing to the Linux Audio spirit around Hilversum/Utrecht, to Miller Puckette for sharing a great piece of software called Pure Data, to IOhannes Zmölnig for offering a tutorial about extending the Pure Data core, to Hans-Christoph Steiner for the Makefile template as well as to all the cited researchers for their effort.

References

J. Murray Barbour. 2004. *Tuning and Temperament*. Dover Publications, Inc., dover edition.

Emilios Cambouropoulos. 2003. Pitch spelling: A computational model. *Music Perception*, 20(4).

Elaine Chew and Yun-Ching Chen. 2005. Real-time pitch spelling using the spiral array. *Computer Music Journal*, 29(2).

A. D. Fokker and Balth. van der Pol. 1942. *Harmonische Muziek*. Martinus Nijhoff, 's-Gravenhage.

Donald J. Grout and Claude V. Palisca. 1988. *A History of Western Music*. W. W. Norton & Company, Inc., fourth edition.

Hermann von Helmholtz. 1896. *Die Lehre Von Den Tonempfindungen Als Physiologische Grundlage Für Die Theorie Der Musik*. Druck und Verlag von Friedrich Vieweg und Sohn, Braunschweig, fifth edition. Replica.

A. K. Honingh. 2006. *The Origin and Well-Formedness of Tonal Pitch Structures*. Ph.D. thesis, Universiteit van Amsterdam. `http://staff.science.uva.nl/~ahoningh/publicaties/proefschrift.pdf`.

H. C. Longuet-Higgins and M. J. Steedman. 1971. On interpreting bach. `http://aitopics.org/sites/default/files/classic/Machine%20Intelligence%206/MI6-Ch15-LonguetHigginsSteedman.pdf`.

Alfred Mann. 1987. *The Study of Fugue*. Dover Publications, Inc., New York, dover edition.

David Meredith. 2003. Pitch spelling algorithms. In *Proceedings of the 5th Triennal ESCOM Conference*.

Funs Seelen. 2014. A parametric dynamic tuning system for the diatonic scale / towards five-limit just intonation based on a hexachord analysis algorithm. *Berichte der Reihe Musikinformatik und Medientechnik*, (52).

Karl L. Signell. 1986, 1977. *Makam - Modal Practice in Turkish Art Music*. Da Capo Press, New York.

David Temperley. 2004. *The Cognition of Basic Musical Structures*. MIT Press, Cambridge, Massachussetts, first paperback edition.

towards message based audio systems

Winfried Ritsch
Institute of Electronic Music and Acoustics Graz
Inffeldgasse 10
8010 Graz,
Austria,
ritsch@iem.at

Abstract

The deployment of distributed audio systems in the context of computermusic and audio installation is explored in the paper, expanding the vision of static streaming audio networks to flexible dynamic audio networks. Audiodata is send on demand only. Sharing sources and sinks allows us arbitrary audio networks.

This lead to the idea of message based audio systems, which has been investigated within two use cases: Playing on an Ambisonics spatial audio system, and within a computermusic ensemble.

In a first implementation Open Sound Control (OSC) is used as the content format proposing a definition of Audio over OSC (*AOO*).

Keywords

audio-interfaces, networked audio, OSC, computermusic ensembles, sound installation

1 Introduction

The first idea of a message based audio system came up with the requirement of playing a multi-speaker environment of distributed networked embedded devices from several computers, avoiding a central mixing desk.

Another demand for a message based audio network came up during the development of a flexible audio network within the *ICE*-ensemble[1][IEM, 2011]. A variable number of computermusic musicians sending time bounded audio material with their computers to other participants (for monitoring or collecting audio material), would have caused a complex audio-matrix setup of quasi-permanent network connections with all the negotiations and initializations for these streams. Not only because of the limited rehearsal time, this seems to be both too error prone and an overkill in terms of network load.

The structure of a functional audio-network for ICE, especially during improvising sessions, cannot always be foreseen and is therefore hard to implement as a static network. It is therefore important to be able to easily change the audio network during performance, as musicians come and leave (and reboot). On the other hand, the need for low latency, responsiveness and sufficient audio quality has to be respected even during the dynamic change of network connections. No strict requirements on sample-rates, sample-accurate synchronization and the use of unique audio formats should be made in such situations. It should be possible to freely add or remove audio related devices to/from the system without having to go through complicated setup of audio streams and without having to negotiate meta data between the participants. This should simplify the implementation of the particular nodes.

Of course, special care has to be taken when playing together in an ensemble. Factors like network overload, especially peaks, can lead to bad sound and feedbacks. On the other hand, we also find such situations when playing together in the analog world. In any case, the limits have to be explored during rehearsals.

Setting up continuous streams where audio data, including silence, is send continuously to all possible destinations is an overhead, that can easily touch the limits of available network bandwidth. But also can cause wasteful/costly implementations. If we can send audio from different sources to sinks (like speaker systems) only on demand, simplifies the setup. Also, reducing the needs for negotiation for establishing connections simplifies this task, and therefore stabilizes the setup.

The use of messages for the delivery of audio-signals in a network seems to contradict the usual implementation of real-time audio-processing implementations in digital audio workstations, where mostly continuous synchronized audio streams are used. If these audio messages are send repeatedly in such a way that

[1]IEM (Institute of Electronic Music and Acoustics) Computermusic Ensemble

they can be combined together in time, they can been seen as limited audio data streams and supersede continuous audio streams.

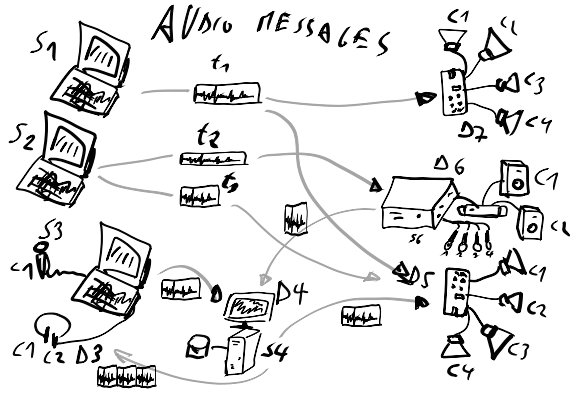


Figure 1: first idea of a message audio system with sources S_n and drains D_n

Summing up these demands, the overall vision is to implement a distributed audio network, where a variable amount of nodes act as sound sources and sound sinks (drains). It should be possible to send audio messages from any source to any sink, from multiple sources simultaneously to a single sink, respectively broadcasting audio messages from one source to multiple sinks. Accordingly, the cross-linking between the audio components is arbitrary, as shown in figure1.

There should not be a "Before you stream audio, you first have to negotiate and connect with ...", Instead, any participant should be able to just send their audio data to others when needed. The receivers should be able to decide how to handle the audio, depending if they can or want to use them.

Following features can be outlined:

- audio signal intercommunication between distributed audio systems
- arbitrary ad hoc connections
- various audio formats, sample-rates
- synchronization and lowest latency possible
- audio-data on demand only

The most common way of communication within local networks is Ethernet. Therefore "Audio over Ethernet" has become a widely used technique. However, there is roughly only a single approach: Stream based audio transmission, representing the data as a continuous sequence. For audio messages as on-demand packet based streams[2] we found no usable implementation (2009). This lead to the design and implementation of a new audio transmission protocol for the demands shown before. As a first approach, an implementation in user space (on the application layer) without the need of special OS-drivers was intended. This can also be seen as the idea of "dynamic audio networks".

2 Audio over OSC

Looking for a modern, commonly used transmission format for messaging systems within the computermusic domain, we found "Open Sound Control" (OSC) [Wright, 2002]. With its flexible address pattern in URL-style and its implementation of high resolution time tags, OSC provides everything needed as a communication format[Schmeder et al., 2010]. OSC specifications points out that it does not require specific underlying transport protocol, but often uses Ethernet network. In our case this would be UDP in a first implementation but is not limited to these. TCP/IP as transport protocol can also be used, but would make some features obsolete and some more complicated, like the requirement for negotiations to initialize connections. Wolfgang Jäger implemented "Audio over OSC" (*AoO*) within a first project at the IEM [Jaeger and Ritsch, 2009]. This was used in tests and "AUON" (all under one net), a concert installation for network art[3]

2.1 the AoO-protocol

The definition of AoO protocol was made with simplicity in mind, targeting also small devices like microcontrollers:

```
AoO message := "#bundle" timestamp
  <format> <channel> [<channel>,...]

format := "/AOO/drain/<d>/format"
  samplerate blocksize overlap mime-type
  [time correction]

channel := "/AOO/drain/<d>/channel/<c>"
  id sequence resolution resampling <data>

d ... number of drain (integer)
c ... channel number (integer)
data ... audio data (blob)
```

[2] not to be mistaken with "streaming on demand" or UDP packets

[3] performed 17.1.2010 in Medienkunstlabor Kunsthaus Graz see `http://medienkunstlabor.at/projects/blender/ArtsBirthday17012010/index.html`

A AoO message is represented by an OSC-bundle with the obligate timestamp. It contains one format message at the beginning and one or more channel messages.

For the addressing scheme the structure of the resources in network is used as the base. Each device in the network with an unique network-address (IP-number and Port number) can have one or more drains. Each of these drains can have one or more channels. There can be an arbitrary amount of drains, and each drain could have an arbitrary amount of channels. An example address of a channel in an device looks like $/AOO/drain/2/channel/3$, where the third channel of the second drain in the device is targeted. $/AOO$ is the protocol specific prefix.

Like described in "Best Practices for Open Sound Control"[Schmeder et al., 2010], REST (Representational State Transfer) style is used. With its stateless representation each message is a singleton containing all information needed.

In OSC, there is a type of query operators called address pattern matching. These can be used to address multiple channels or drains in one message. Since pattern matching can be computational intensive, we propose only to use the "*" wild-char for addressing all channels of a drain or all drains of a device. For instance the channel message $/AOO/drain/2/channel/*$ will use the audio data for all channels of the second drain.

A OSC format message, with for example $/AOO/drain/2/format$ as address string, is always the first item in the bundle and specifies the samplerate, the blocksize and overlap factor as integer, followed by a string as the mime-type of the audio data. The optional time correction factor will be explained at section 2.3.

Integer was chosen in favor for processors without hardware floating point support. Channel specific data information like the id number of the message stream, the sequence number in the channel message allow more easily to detect lost packages. The resolution of a sample and an individual resampling factor is contained in the channel messages, where the resampling factor enables channels to differ from the samplerate specified in the format message, allowing lower rates for sub channels, control streams or higher rates for specific other needs. This also becomes handy if FFT-frames are transmitted as data.

Some of the header data is shown in the following summary example to explain some specific features of the audio transmission:

sampling rate Different sampling rates of sources are possible, which will be resampled in the drain.

blocksize The amount of samples in each package of audio data, which must be greater or equal 1, limited by packet size.

overlapping factor The overlapping factor is 1 (one) by default. Higher values can be used to implement redundancy, to deal with lost packets or needed when sending FFT-frames (in future implementations).

resampling factor is linked to the samplingrate in order to be able to choose the precision of each channel individually using oversampling or similar.

coding of the audio data using the *Multipurpose Internet Mail Extensions* (MIME) standard[Authority, 2009]. In our uncompressed format, the MIME type would be "audio/pcm", whereas "audio/CELP" classifies CELP encoded data.

In order to send usable data, sources have to be aware of the formats a given drain can handle. [4]

data types preferred are uncompressed packets with data types defined by OSC, like 32-Bit float. However, it's also possible to use blobs with an arbitrary bit-length audio data. This can become handy if bandwidth matters. Sources must be aware, which formats can be handled by the drains.

To provide low latency, time-bounded audio transmissions should be sliced into shorter messages and send individually to be reconstructed at the receiver.

2.2 drains

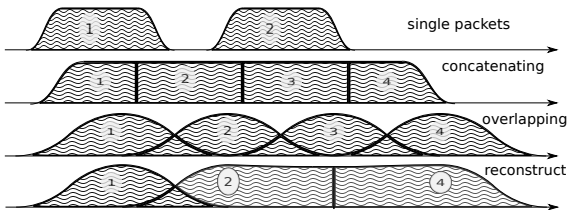


Figure 2: audio messages are arranged as single, combined or overlapped

Sending audio data is simply slicing and adding timestamps. On the other side, receiving

[4]This subject is currently under discussion, and may get changed in the future

means that audio packets have to be rescheduled and synchronized on the time-line, using the timestamp, sequence and id received, and mixed together. Mixing is required either if audio packets come from different sources, have different ids or if they are overlapping (using an overlapping factor greater than one). Audio messages also have to be re-sampled before they are added, to handle with sources with different samplerates. Even if nodes are using the same nominal sample-rate, they are usually not sample-synchronized, since the sample-clocks can drift in time. The re-sampling factor can therefore change dynamically.

For re-arranging the audio packages there is a need to do some sort of labeling of the messages, since it is not clear if they are intended to overlap or are different material. This can be handled via the "identification number" (id). Identical identification numbers means to recognize the material as one material and they can be cross-faded. So these numbers has to has to unique at least at the drain.

The first audio packet has to be faded in and the last faded out. A sequence of audio messages must be concatenated. At least one message has to be buffered to know if a next one arrives. If messages are in overlapping mode, they always have to be cross-faded. If one packet is lost in the overlapping mode, the signal can be reconstructed.

2.2.1 addressing problems

Like described above, to deliver audio messages to a drain, additionally to the drain number and channel number, the address of the device has to be known. A decision was made, that the address is not part of the message, since the sender has to know about the drain on the receiver and the network system has to handle the addressing. Since automatic IP assignment can be used, this could make the argument to simplify the network obsolete, since we devices have no static address.

Like stated in in the vision, we do want negotiations and requests, but in situations where IPs are unknown, we needed a mechanism to grasp it. One implementation was announcement message broadcasted by each drain, with the address and a human readable meaningful name. Even more polite we implemented them as invitation messages, which also states: "ready to receive". This was done every 10 seconds, to limit load and also serves as a live message.

A second problem arose, since broadcasting to all drains with the same number, the destination information is not contained in the audio message, we cannot use broadcast to reduce network load and address specific destinations. For this the drain has to know about the sources it will accept. Anyway this worked fine, but made some additional efforts in communication before.

Anyway addressing is in heavy discussion, has to be tested further on use cases and will probably change in future.

2.2.2 mixing modes

In this first implementation we used two different modes: Mode 1 provides the possibility of summation of the received audio signals and Mode 2 should perform an arithmetic averaging of parallel signals. The reason for this is that summing audio signals with maximum amplitudes each causes distortion. Using Mode 2 this cannot happen. If many sources play into one drain, this can also be seen as a kind of mix to reduce the impact of a single one. Sometimes automatic level control or limiting in the digital domain after adding the signals is the better way to prevent clipping.

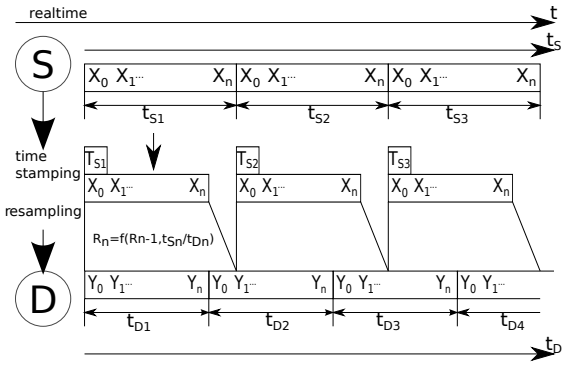


Figure 3: re-sampling rate R_n between source S and drain D is not constant

2.3 timing and sample-rates

Timing is critical in audio-systems, not only for synchronizing audio, but also to prevent jitter noise. Time tags of the packets are represented by a 64 bit fixed point number, as specified by OSC, to a precision of about 230 picoseconds. This conforms to the representation used by the Network Time Protocol (NTP)[Mills et al., 2010].

In fixed buffering mode, the buffer size has to be chosen large enough to prevent dropouts. In the automatic buffer control mode, the drain

should use the shortest possible size for buffering. If packets arrive too late, buffering should be dynamically extended and then slowly reduced.

Since audio packets can arrive with different sample-rates, re-sampling is executed before the audio data is added to the internal sound stream synchronized with the local audio environment. This provides the opportunity to synchronize audio content respecting the timing differences and time drifts between sources and drains. This strategy of resampling is shown in figure 3:

Looking at synchronization in digital audio system, mostly a common master-clock is used for all devices. Since each device has its own audio environment, which may not support external synchronization sources, the time T_Sn of the local audio environment is used to calculate the timestamp for outgoing audio messages.

Using the incoming timestamps from the remote source, we can compare them with the local time t_Dn and correct the re-sampling factor R_n dynamically for each message. The change of the correction should be small if averaged over a longer time, but can be bad for first audio messages received.

For a better synchronization of audio data, a Time Transfer protocol can be used in parallel to synchronize the drain with the source, as a sort of master-clock.

Therefore, as proposed in the OSC specifications, NTP can be used for each node. Another time protocol for synchronization of audio data is the Precision Time Protocol (PTP)[on Sensor Technology, 1588–2002], e.g. also used in AVB[5], allows a more lightweight implementation in local networks and can guarantee a quasi sample-accurate synchronization. PTP is the preferred time protocol to be used with AoO. For these protocols we need a master (or grandmaster) in the network. This is done differently depending on the used implementation of the time protocol.

Since the local time source of a device can differ from the timing of the audio environment, each device needs a correction factor between this time source and the audio hardware time including the time master device. This factor has to be communicated between the devices, so the re-sampling correction factor can be calculated before the first audio message is sent, guaranteeing a quasi sample-synchronous network-wide system starting with the first message send.

2.4 Requests

Asking won't hurt. If the drain provides information about its capabilities, it can be used to optimize and ensure the transmission. However, this information is optional, allowing simple implementations on some nodes, like microcontrollers, that may be unable to accomplish this task. Until now there is no proposal how to implement such requests, instead we used announcement/invitation messages for grasping the sources in the local net.

2.5 Implementation

As a first proof of concept, *AoO* was implemented within user space using Pure Data[Puckette, 1996]. This implementation has shown various problems to be solved in future. Using the network library iemnet[6] additional "externals" have been written in C to extend the OSC-Protocol, split continuous audio signals into packets and mix OSC audio messages in drains. As repository for the GPL open source can be found at the "Opensource@IEM" sourceforge as git repository site at:

`http://sourceforge.net/p/iem/aoo/`

As a first test environment, a number of different open-source audio hardware implementations, using Debian Linux OS-System, has been used. The test patches were written with Pd version 0.42.5, where a central component has been the OSC library of Martin Peach. Later, an implementation for a microcontroller board "escher2"[Algorythmics, 2012] has been created, which has been superseded by small embedded arm-devices, like beaglebones[Foundation, 2013], also using a Debian OS system.

3 message based Ambisonics spatial audio systems

As a first goal, the geodesic sound-dome in Pischelsdorf (with a diameter of 20 m and a height of about 10 m) as an environmental landscape sculpture in Pischelsdorf should transmute into 3D a sound-sphere. Therefore as special hardware and software, a low power solar power driven multichannel Ambisonics system was developed and installed prototypically. This should result in a low cost implementation of multichannel audio system Up to 48 speakers

[5] Audio Video Briding, Standard IEEE 802.1

[6] iemnet project site is `http://puredata.info/downloads/iemnet`

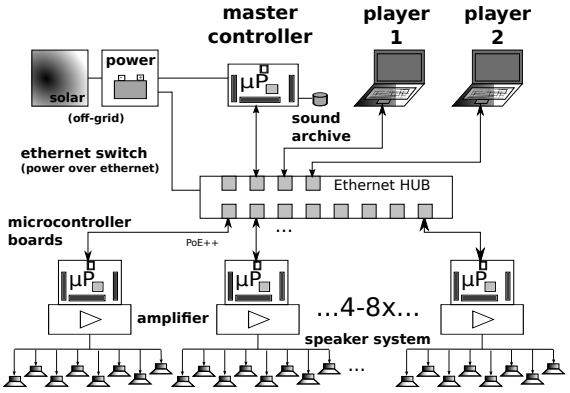


Figure 4: *AoO* with embedded devices for spatial audio system

should be mounted in a hemisphere, forming an Ambisonics sound system. Using 6 nodes, each with 8 speakers, special embedded controllers are used to render the audio in the system (figure 4).

Figure 5: One node with one speaker in the dome

Each node is a small embedded computer equipped with an 8-channel sound-card, including amplifiers and speakers. Each speaker can been calibrated and fed individually. However, since each unit is aware of its speaker positions, it can also render the audio with an internal Ambisonics encoder/decoder combination.

So instead of sending 48 channels of audio to spatialize one or more sources, the sources can be broadcast combined with OSC-spatialization data and the drains render them independently. Another possibility is to broadcast an encoded Ambisonics-encoded multichannel signal, where the devices decode the Ambisonics signal for their subset of speakers. The Sound Environment can be sent from one master controller or any other connected computer.

The first implementation of the nodes has been done with special micro-controller boards *escher2*[Algorythmics, 2012] which drive the custom designed DA-Amp boards. Since these devices have very limited memory (max. 16 samples of 64 channels), standard Linux audio system cannot provide the packets small and fast enough for a stable performance without special efforts, like own driver in kernel space for the packet delivery. Therefore a major problem has been the synchronization and the reliability of the transmission, but providing latency.

One other solution could be, to secure resources like bandwidth and computing time with restricting audio data to be sent on defined time slots: only and one time-slot for each device. Most of the available network-components are able to handle the IP-protocol or even OSC but unfortunately there is no commonly used computer OS, which is able to deliver audio data in dedicated time-slots. Therefore as one implementation of hard real-time networking for real-time Linux, the RTnet[Team, 2002] has been found. It needs a hard-realtime kernel. In a further thought the OSC-Transmission has to be implemented as a Linux-device, coupling with the RT-Net Ethernet driver.

Since 2012 small embedded Linux-systems like the beaglebone black[Foundation, 2013] are available and can be used to drive the DAs with amplifiers. This has been tested recently with good success on a beaglebone black: An acceptable latency of 5-10 ms with 8 out-channels has been achieved .

Figure 6: sounddome as hemisphere, 20 m diameter in cornfield

The main advantage, besides the low cost and autonomous system, is that one or more sound technicians or computer musicians can enter the

dome, plug into the network with their portable devices and play the sound dome either addressing speakers individually, with audio material spatializing live with additional OSC messages or a generated or prerecorded Ambisonics audio material.

3.1 Playing together

Figure 7: first concert of IEM computermusic ensemble ICE playing over a HUB

When specifying an audio-network for playing togehter within an ensemble, a focus was set on the collaborating efforts to be done to gain the unity of the individuals.

So, like a musicians with acoustic instrument, joining a band with Linux audio-computer implies a need for a place where the musician has a "virtual sound space" they can join. So they provide sound sources and need to plugin audio channels on a virtual mixing desk. With *AoO* the participant just needs to connect to the network, wireless or wired, choosing the drains to play to and send phrases of audio with AoO when needed.

For the ICE ensemble Ambisonics as an virtual audio environment was chosen, which can be rendered to different concert halls. Within the Ambisonics each musician can always use the same playing parameters for spatializing her or his musical contribution. So the imagination of the musician is "playing in a virtual 3D environment", sending their audio signals together with 3D-spatial data to a distributed mixing system which is rendering it on the speakers.

Additional there is an audio communication between the musicians, where each musicians can hear into the signal produced by the other, if there is one or on special offered drains send audio intervention to the others for e.g. monitoring purposes. The musicians can do their own monitor mix, depending on the piece and space where the play.

Using a message audio system, each musicians only sends sound data if playing, like audio bursts just notes, or just sending their audio-data to another musicians, who will process this further and so on. There should be no border on the imagination of these situations, (as long it can be grasped by the participants).

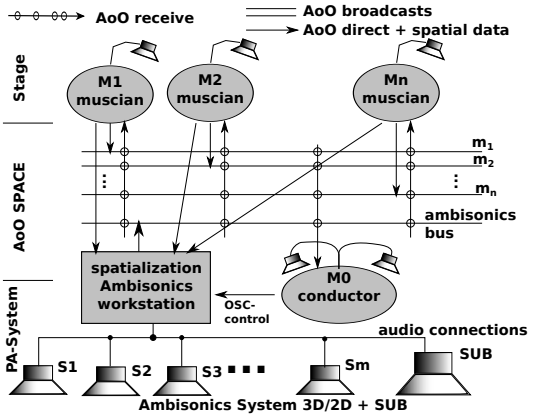


Figure 8: ICE using AoO as space for playing together and on a PA system6

4 state of the work

The *AoO* has been implemented for proof of concept and special applications in a first draft version. The next version should fixate the protocol, after having discussed it in public, in a way that makes it compatible with future protocol upgrades.

The usage of *AoO* in an ensemble has been explored in a workshop with students at the IEM, but the implemented software was not stable enough on the different platforms used for stage performance. This was especially true, when we tried to reach the short latencies needed for concerts. Some more programming efforts has to be done, to guarantee better timing using different computer types, within different Linux-implementations and setups.

Running *AoO* on embedded Linux devices has shown to be successful, if the devices are tweaked for real-time audio usage. The development on the *escher2* (dsPIC33E-)microcontroller board has been abandoned in favor of the new generation of small low power embedded devices with arm processors. A first version of implementation (V1.0) of *AoO* is scheduled for April 2014 for a public installation in the sound-dome, where the Ambisonics audio-system should be finalized for permanent perfor-

mance and open access. More documentation and source code should be released and open-hardware as *AoO*-audio devices should be available.

Special focus is done on using embedded devices with *AoO* as networked multichannel audio hardware interfaces for low cost solutions adding audio processing for calibration filters, beam-forming,... for speaker-systems optional powered over Ethernet.

5 Conclusions

Starting as a vision, these experiments and implementations have shown, that message based audio systems can enhance the collaboration in ensembles, playing open audio systems. Also network art projects using the Internet can use *AoO* to contribute to sound installation from outside, just knowing the IP and ports to use.

The implementation is far from being complete, and more restrictions will be included in order to simplify the system. Synchronization and re-sampling is not perfect, but usable for most cases and it has been shown, that audio message systems can work reliable in different situations.

Audio message systems can also be implemented in other formats than OSC and lower layers of the Linux OS, like jack-plugins or ALSA-modules as converters between message based audio system and synchronous data flow models.

For really low latency (below 1 ms) using *AoO* as audio over Ethernet system, kernel-drivers must be developed and with time-slotted Ethernet transmissions, systems with latencies down to 8 us on transmission time can be implemented using hard RT-systems.

6 Acknowledgements

Thanks to ... my colleagues on the IEM supporting me with their help, especially Wolfgang Jäger for a first implementation as a sound-engineering project. Also for helping set up the "Klangdom" especially to Marian Weger, Matthias Kronlachner and the cultural initiative K.U.L.M. in Pischelsdorf and the members of the ICE Ensemble helping to experiment and many others. Thanks also for corrections of this paper and useful hints, to enhance the understanding.

References

Atelier Algorythmics. 2012. escher development. `http://algo.mur.at/projects/microcontroller/escher`. [Online; accessed 1-Feb-2014].

Internet Assigned Numbers Authority. 2009. MIME Media Types. `http://www.iana.org/assignments/media-types/`. [Online; accessed 1-Feb-2014].

The BeagleBoard.org Foundation. 2013. beaglebone black. `http://beagleboard.org/products/beaglebone\%20black`. [Online; accessed 1-Feb-2014].

Winfried Ritsch IEM. 2011. IEM Computer Music Ensemble. `http://iaem.at/projekte/ice`. [Online; accessed 1-Feb-2014].

Wolgang Jaeger and Winfried Ritsch. 2009. AOO. `http://iem.kug.ac.at/en/projects/workspace/2009/audio-over-internet-using-osc.html`. [Online; accessed 12-Dez-2011].

D. Mills, J. Martin, J. Burbank, and W. Kasch. 2010. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June.

Technical Committee on Sensor Technology. 1588–2002. *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. The Institute of Electrical and Electronics Engineers, Inc. (Hrsg.), New York, ieee std. edition, November.

M. Puckette. 1996. Pure Data. In *Proceedings, International Computer Music Conference.*, page 224–227, San Francisco.

Andrew Schmeder, Adrian Freed, and David Wessel. 2010. Best Practices for Open Sound Control. In *Linux Audio Conference*, Utrecht, NL, 01/05/2010.

RTnet Development Team. 2002. RTNET. `http://rtnet.org/`. [Online; accessed 1-Feb-2014].

Matt Wright. 2002. The open sound control 1.0 specification. `http://opensoundcontrol.org/spec-1\_0`. [Online; accessed 1-Feb-2014].

The JamBerry - A Stand-Alone Device for Networked Music Performance Based on the Raspberry Pi

Florian Meier
Hamburg University of Technology
Schwarzenbergstraße 95
21073 Hamburg, Germany,
florian.meier@tuhh.de

Marco Fink and **Udo Zölzer**
Dept. of Signal Processing
and Communications
Helmut-Schmidt-University
Holstenhofweg 85,
22043 Hamburg, Germany,
marco.fink@hsu-hh.de

Abstract

Today's public Internet availability and capabilities allow manifold applications in the field of multimedia that were not possible a few years ago. One emerging application is the so-called Networked Music Performance, standing for the online, low-latency interaction of musicians. This work proposes a stand-alone device for that specific purpose and is based on a Raspberry Pi running a Linux-based operating system.

Keywords

Networked Music Performance, Audio over IP, ALSA, Raspberry Pi

1 Introduction

The ways of today's online communication are versatile and rapidly evolving. The trend went from text-based communication, over audio-based communication, and finally constituted in multimedia-based communication. One arising branch of online communication is the so-called Networked Music Performance (NMP), a special application of Audio over IP (AoIP). It allows musicians to interact with each other in a virtual acoustic space by connecting their instruments to their computers and a software-based link-up. This procedure allows artistic collaborations over long distances without the need of traveling and hence, can enrich the life of artists. Instead of increasing the content dimensionality and therefore the data rate, the challenge in AoIP is to fulfill a certain delay threshold that still allows musical interaction.

For the purpose of providing an easy-to-use system realization, an all-in-one device, entitled the *JamBerry*, is presented in this work. The proposed system, as shown in Fig. 1, consists of the well-known Raspberry Pi [1] and several custom hardware extensions. These are necessary since the Raspberry Pi does not provide high-quality audio output and no audio input at all. Furthermore, the proposed device includes several hardware components allowing a quick and simple connection of typical audio hardware and instruments. The Raspberry Pi itself can be described as chip-card-sized single-board computer. It was initiated for educational purposes and is now widely-used, especially in the hardware hobbyist community since it provides various interfaces for all sorts of extensions.

Figure 1: The JamBerry Device

The paper is structured as following. An introduction into the topic of Audio over IP is given in Section 2, including the requirements and major challenges when building such a system. Section 3 gives a detailed view on the actual AoIP software running on the JamBerry. The necessary extensions of the Linux audio drivers and the integration in the ALSA framework is depicted in Section 4. The custom hardware extensions to the Raspberry Pi are explained in Section 5. Section 6 highlights the capabilities of the JamBerry in the contexts of audio and network parameters, whereas concluding thoughts can be found in Section 7.

2 Audio over IP

Transmission of Audio over IP-based networks is nowadays a wide-spread technology with two main applications: Broadcasting audio streams and telephony applications. While the first one provides no return channel, the second one allows for direct interaction over large distances. Although, the requirements in terms of audio

quality and latency for playing live music together are not fulfilled by current telephony systems.

The massive spreading of broad-band Internet connections and increase in network reliability allows the realization of AoIP systems now. Therefore, this topic gained much research attention in the last years. A good introduction into the topic of Networked Music Performances and the associated problems can be found in [2], while [3] gives an extensive overview of existing systems.

An early approach was SoundWIRE [4] by the Center for Computer Research in Music and Acoustics (CCRMA), where later JackTrip [5] was developed. JackTrip includes several methods for counteracting packet loss such as overlapping of packets and looping of data in case of a lost packet. It is based on the JACK sound system, just like NetJack [6] that is now part of JACK itself. To avoid the restriction to JACK, Soundjack [7] is based on a generic audio core and hence, allows cross-platform musical online interaction.

The Distributed Musical Rehersal Environment [8] focuses on preparing groups of musicians for a final performance without the need to be at the same place. Remote rehersal is also one of the applications of the DIAMOUSES framework [9] that has a very versatile platform including a portal for arranging jam sessions, MIDI support and DVB support for audience involvement.

2.1 Requirements

The goal of this project was to build a complete distributed music performance system to show the current state of research and establish a platform for further research. The system is supposed to be usable in realistic environments such as rehearsal rooms. Therefore, it should be a compact system that integrates all important features for easy to setup jamming sessions. This includes two input channels with various input capabilities to support high-amplitude sound sources such as keyboards or preamplified instruments, as well as low-amplitude sound sources like microphones and passive guitar pickups. Furthermore, it should drive headphones and provide line-level output signals.

The system should support sampling rates of 48 kHz with a bit depth of 16 bit. Higher values do not provide much benefit in quality. Furthermore, no further signal processing steps, depending on highly-detailed signaled representations, are involved. To allow the interaction with several musicians but still stick to the computational constraints of the Raspberry Pi, the system shall support up to four interconnected JamBerries.

2.2 Challenges

Transmission of audio via the Internet is considerably different from point-to-point digital audio transmission techniques such as AES/EBU [10] and even Audio over Ethernet (AoE) techniques like Dante or EtherSound [11]. The transmission of data packets via the Internet is neither reliable nor properly predictable. This leads to audio data being considerably delayed or even vanished in the network.

This is commonly counteracted by using large data buffers where the packets arriving in irregular time intervals are additionally delayed so that late packets can catch up. Unfortunately, large buffers are contradictory to the requirements of distributed music performances since a minimum latency is essential. Interaction of several musicians is solely achievable when the round trip delay does not exceed a certain threshold [12; 13]. Secondly, even large buffers do not prevent dropouts resulting from lost packets. Therefore, this project takes two completive approaches:

- Audio data packets that do not arrive in time are substituted by a technique called error concealment. Instead of playing back silence, audio is calculated from preceding data.
- The data buffer length is dynamically adjusted to the network conditions. This enables minimum latency while still providing good audio quality.

3 Software System

The AoIP software itself is a multi-threaded C++11 application running in user space. It accesses the audio hardware via the well-known ALSA [14] library. The user interaction takes place via a WebSocket interface that enables the use of a JavaScript/HTML GUI that can be accessed via the integrated touchscreen as well as from a remote PC or tablet. The WebSocket interface is provided by a library [15] written during this project running the WAMP [16] protocol.

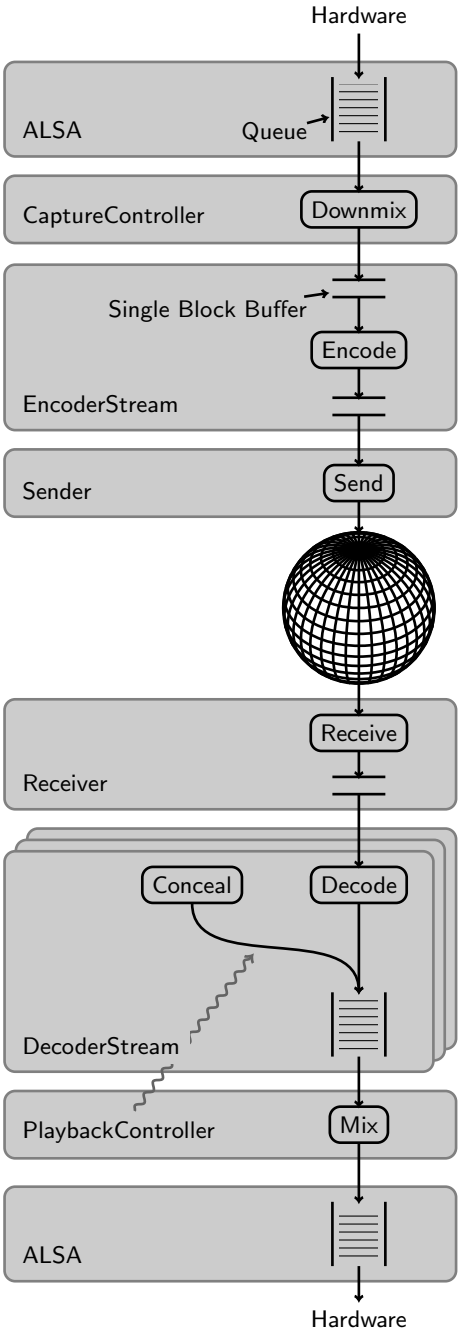


Figure 2: Data flow of the JamBerry software

The data flow of the audio through the software is depicted in Fig. 2. Audio is captured from the hardware via the ALSA library. As soon as a block (120 or 240 samples) of new data is available, it is taken by the *CaptureController* that mixes the signal down to a single channel. Transmitting multiple streams is possible, too, but provides a negligible benefit in this scenario. The data can be transmitted as raw data. Alternatively, the required data rate can be reduced by utilization of the Opus [17; 18] low-latency coding procedure. The encoding is done by the *EncoderStream* that passes the data to the sender for sending it to all connected peers via unicast UDP. Currently, there is no discovery protocol implemented, so the peers have to be entered manually. As soon as the data is received at the receiver, it is decoded and pushed into the receiver buffer queue. The *PlaybackController* mixes the data from various sources and enables ALSA to access the result. Thus, a continuous reading of data is realized. In the case of missing data an error concealment procedure is triggered to replace the missing data and avoid gaps in the playback. The current implementation utilizes the concealment procedure from the Opus codec, since its complexity is low in contrast to other known concealment strategies [19; 20; 21]. Alternatively, the last block can be repeated until newer data arrives (so-called "wavetable mode" as in [5]). The queuing process at the receiver is explained in more detail in the following.

3.1 Adaptive Queuing

In order to achieve minimum latency while maintaining good audio quality, the length of the audio buffer is adjusted to the network conditions within the playback thread. The corresponding control routine is depicted in Fig. 3.

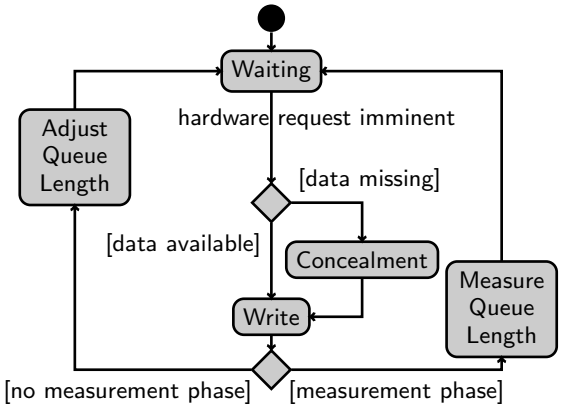


Figure 3: Process of Playback Thread

The ALSA data queue is kept very short to avoid unnecessary delays that would increase the overall latency. The *PlaybackController* monitors the state of ALSA and just before the hardware will request new data, it is written to the ALSA buffer. Whenever current audio data exists in the moment of the hardware request, this data is utilized. In the case of missing data, the error concealment routine is triggered to produce the corresponding data. The computa-

tion of concealment data takes some time. This period of time is taken into account to provide the data just at the right point in time.

In order to maintain a reasonable buffer size, a simple open-loop control was implemented. A buffer size that is unreasonably large would result in useless delay. When the buffer is too small, a major part of the audio packets arrives too late. Although a certain amount of packets can be concealed, the audio quality decreases with a rising amount of lost packets.

Right after a new connection was established, the buffer size is set to a very high value. In the following few seconds, the length of the queue in samples Q is measured and the standard deviation σ_Q is calculated. After the measurement phase is over, the optimal queue length is calculated as

$$Q_{opt} = \beta \cdot \sigma_Q, \tag{1}$$

where the constant $\beta \geq 1$ accounts for packets outside the range of the standard deviation. When the current queue length is outside the interval

$$[Q_{opt} - Q_{tol}, Q_{opt} + Q_{tol}], \tag{2}$$

the corresponding number of samples is dropped or generated. Once the queue is adjusted to the current network characteristic, this occurs very infrequently so the audible effect is insignificant. The parameters β and Q_{tol} are used to trade-off the amount of lost packets, and therefore the audio quality, against the latency.

4 Linux Kernel Driver

The Raspberry Pi has neither audio input nor proper audio output. The existing audio output is driven by a pulse-width modulation (PWM) interface providing medium quality audio. Fortunately, there is another possibility for audio transmission: The Broadcom SoC on the Raspberry Pi provides a digital I^{2}S interface [22] that can be accessed by pin headers. Together with an external audio codec as explained in the next section, this enables high quality audio input and output. However, the Linux kernel lacked support for the I^{2}S peripheral of the Raspberry Pi. An integral part of this project was therefore to write an appropriate kernel driver.

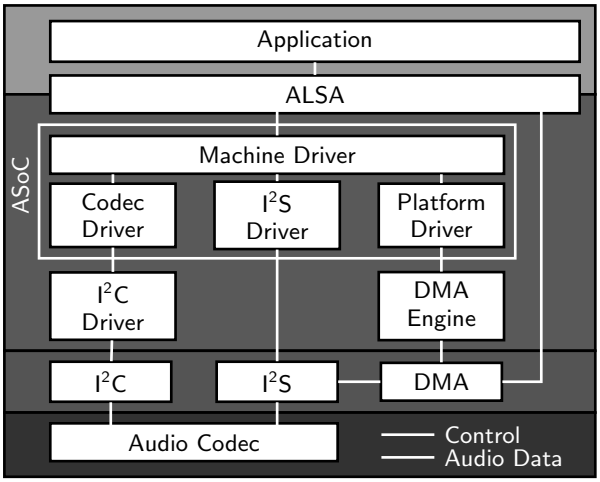


Figure 4: Structure of ASoC and the embedment into the Linux audio framework

Since this driver should be as generic as possible, it is implemented as a part of the ALSA System on Chip (ASoC) framework. It is a subsystem of ALSA tailored to the needs of embedded systems that provides some helpful abstractions that makes it easy to adapt the driver for use with other external hardware. Actually, today there is quite a large number of both open and commercial hardware that uses the driver developed during this project.

Fig. 4 depicts the general structure of ASoC as used for this project. When an application starts the playback of audio, it calls the corresponding function of the ALSA library. This again calls the appropriate initializers for the involved peripheral drivers that are listed in the machine driver. In particular this is the codec driver that is responsible for control commands via I^{2}C, the I^{2}S driver for controlling the digital audio interface, and the platform driver for commanding the DMA engine driver. DMA (Direct Memory Access) is responsible for transmitting audio data from the main memory to the I^{2}S peripheral and back. The I^{2}S peripheral forwards this data via the I^{2}S interface to the audio codec. For starting the playback of the codec, the codec driver will send an appropriate command by using the I^{2}C subsystem. The codec driver is used for transmitting other codec settings such as volume, too.

These encapsulations and generic interfaces are the reason for the software structure's flexibility and reusability. For using a new audio codec with the Raspberry Pi, only the codec driver and the slim machine driver have to be replaced. In many cases only the wiring by the machine driver has to be adapted since there are already many codec drivers available. The spreading of these drivers is based on the frequent usage of ASoC on different platforms.

5 Hardware

Since the Raspberry Pi does not provide proper analog audio interfaces, major effort was spent designing audio hardware, matching the NMP requirements. Furthermore, a touchscreen for user-friendly interaction was connected that requires interfacing hardware. Due to these extensions, the JamBerry can be used as a standalone device without the need of external peripherals such as a monitor.

An overview of the external hardware is depicted in Fig. 5. The extension's functionality is distributed module-wise over three printed circuit boards: A codec board that contains the audio codec for conversion between analog and digital domain. It is stack mounted on the Raspberry Pi. This board is connected to the amplifier board that contains several amplifiers and connectors. The third board controls the touchscreen and is connected to the Raspberry Pi via HDMI. In the following, the individual boards are explained in more detail.

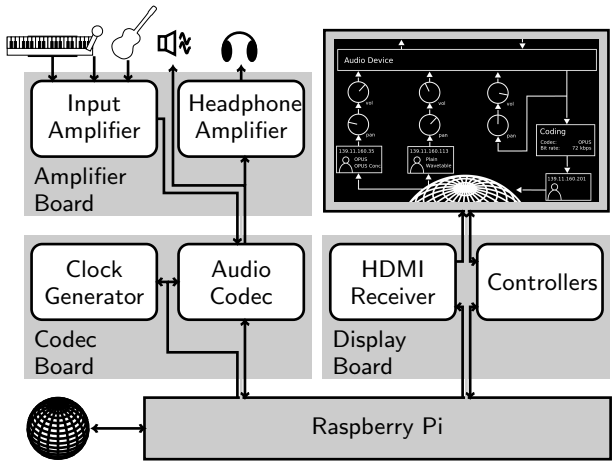


Figure 5: Hardware Overview

5.1 Codec Board

The main component on the digital audio board is a CS4270 audio codec by Cirrus Logic. It has integrated AD and DA converters that provide sample rates of up to 192 kHz and a maximum of 24 bits per sample. It is connected to the I^2S interface of the Raspberry Pi for transmission of digital audio and to the I^2C interface for control. A linear voltage regulator provides power for the analog part of the audio codec, while the digital part is directly driven by the voltage line of the Raspberry Pi. The audio codec is controlled by an external master clock generator. This enables fine-grained synchronization of the sampling frequency on different devices and prevents clock drifts as shown in [23]. The MAX9485 clock generator provides this possibility by a voltage controlled oscillator that can be tuned by an external DAC.

5.2 Amplifier Board

The analog audio board is designed to provide the most established connection possibilities. On the input side two combined XLR/TRS connectors allow the connection of various sources such as microphones, guitars or keyboards. Since these sources provide different output levels that have to be amplified to line-level for feeding it into the audio codec, a two-stage noninverting operational amplifier circuit is placed channel-wise in front of the AD conversion unit. It is based on OPA2134 amplifiers by Texas Instruments that have proven their usability in previous guitar amplifier projects. The circuit allows an amplification of up to 68 dB.

On the output side a direct line-level output is provided as well as a MAX13331 headphone amplifier. It can deliver up to 135 mW into 32 Ω headphones. Furthermore, the analog audio board contains the main power supply for the JamBerry.

5.3 Touchscreen Driving Board

In order to provide enough display space for a pleasant usage experience, but still maintain a compact system size, a 7" screen size is used. A frequently used, thus reliable, and affordable resistive touchscreen of that size is the AT070TN92. For using it together with the Raspberry Pi, a video signal converter is needed to translate from HDMI to the 24 bit parallel interface of the TFT screen. This is provided by a TFP401A by Texas Instruments. The touch position on the screen can be determined by measuring the resistance over the touch panel. This measurement is subject to noise that induces jittering and results in imprecise mouse pointers. The AD7879-1W touch controller is used to conduct this measurement since it provides integrated mean and median filters that reduce the jitter and is controlled via I^2C. The same interface is provided by a DAC for controlling the backlight of the TFT. An additional cable connection for the I^2C connection was avoided by reusing the DDC interface inside the HDMI cable as carrier for the touch and brightness information.

6 Evaluation

The system was evaluated in terms of overall latency introduced by the network as well as audio quality.

6.1 Network

In order to evaluate the behavior of the system under various and reproducible network conditions, a network simulator was implemented. Fig. 6 shows the use of a single JamBerry device connected to the simulator that bounces the received data back to the sender.

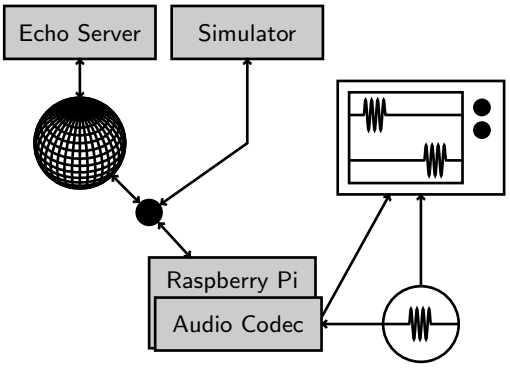


Figure 6: Software Evaluation System

For calibrating the simulator to real conditions a network connection of 13 hops to a server, located in a distance of 450 km, is used. Fig. 7 shows the distribution of the packet delay. The average delay is about 18 ms with a standard deviation of 4 ms.

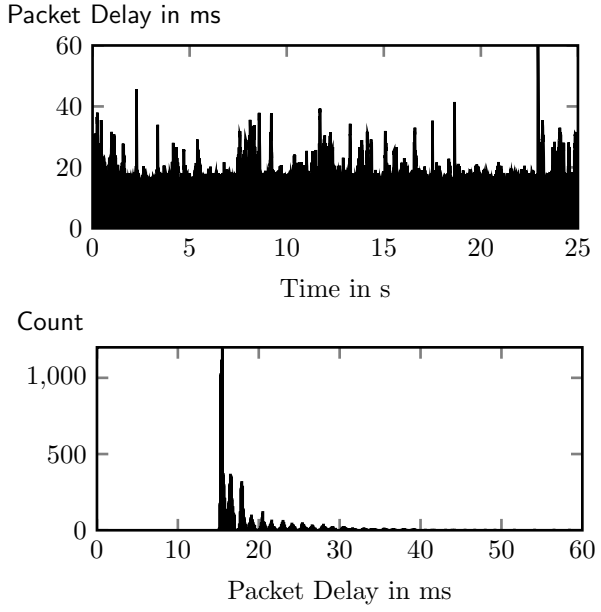


Figure 7: Time series and histogram of the packet delay over the test route

The overall latency is measured by generating short sine bursts and feeding them into the JamBerry. This signal is compared to the resulting output signal by means of an oscilloscope. In addition, GPIO pins of the Raspberry Pi are toggled when the sine burst is processed in different software modules as presented in Sect. 3.

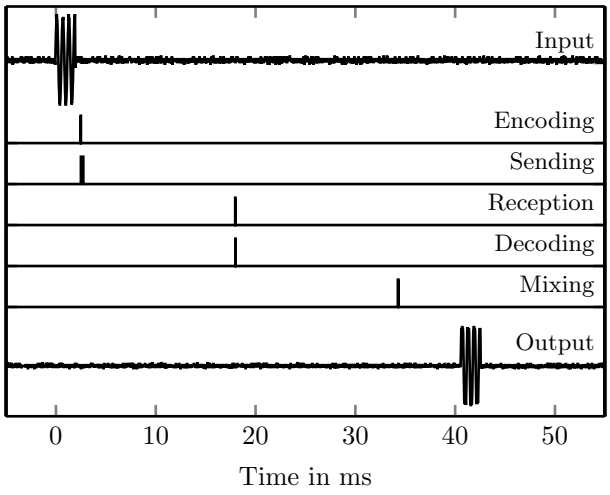


Figure 8: Journey of a Sine Burst

A resulting oscillogram can be seen in Fig. 8. The overall latency is about 40 ms. The time between sending and reception is 15 ms. This matches the time for the actual transmission. Between decoding and mixing, the packet is delayed in the buffer queue for about 16 ms. This buffering is needed to compensate the high jitter of the connection.

For the following evaluation, the overall latency is measured by using the above method while recording the amount of lost packets. Fig. 9 demonstrates the influence of factor β in Eq. (1) while having a constant jitter variance of 9.5 ms^2. With low β, the optimal queue length is short, so the overall latency is short, too. Although, since there is less time for late packets to catch up, the amount of packet loss is very high. With increasing β, the amount of lost packets decreases, but the latency increases. Since sophisticated error concealment algorithms can compensate up to 2% packet loss [19], a constant $\beta = 3$ is chosen for the next evaluation, which is illustrated in Fig. 10. It demonstrates how the control algorithm handles various network conditions. With increasing network jitter variance, the system is able to adapt itself by using a longer queue length. This increases the overall latency, but not the packet loss so the audio quality stays constant.

6.2 Audio

The evaluation of the JamBerry's audio quality was performed module-wise. Therefore, the audio output, audio input, the headphone amplifier and pre-amplifiers were independently inspected. First of all, the superiority of the proposed audio output in contrast to the original PWM output shall be demonstrated.

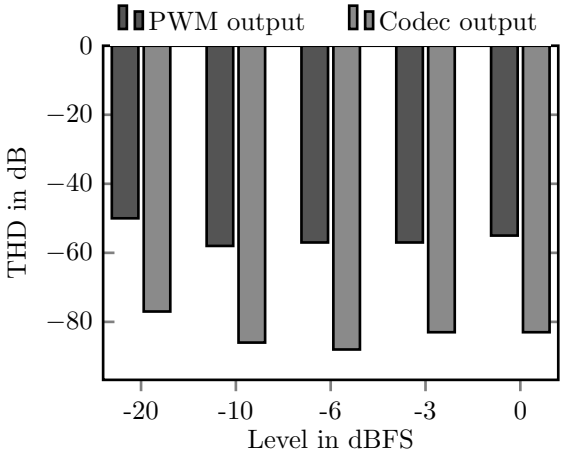


Figure 11: THD of the Raspberry Pi PWM and the codec board output for a 1 kHz sine using different signal levels

If a 1 kHz sine tone is replayed using both outputs accordingly and inspect the corresponding output spectra, as done in Fig. 12, it becomes apparent that the quality is increased significantly using the new codec board. The PWM output introduces more distortion, visible in Fig. 12 in form of larger harmonics at multiples of the fundamental frequency. For example, the amplitude of the first harmonic differs in about 40 dB. Also the noise floor at higher

	Level in dBFS	THD in dB	SNR in dB
Outputs			
PWM output	0	-57	55
Codec output	0	-81	80
Input			
Codec input	0	-91	71

Table 1: Digital audio hardware characteristics

	Gain in dB	THD in dB	SNR in dB
Amplifiers			
Headphone	16	-85	79
Input	17	-81	66
Input	34	-74	48

Table 2: Analog audio hardware characteristics

frequencies is significantly lower. A difference of up to 10 dB can be recognized in Fig. 12. At 50 Hz ripple voltage from the power supply can be seen. Using a power supply of higher quality can reduce this disturbance.

The distortion and noise, audio hardware introduces to audio signals signal is typically expressed in total harmonic distortion (THD) and Signal-Noise-Ratio (SNR), respectively. THD describes, in most conventions, the ratio of the energy of harmonics, produced by distortion, and the energy of the actual signal. In contrast, SNR represents the ratio between the original signal energy and the added noise.

The THD's of the two outputs are illustrated for several signal levels in Fig. 11. Apparently,

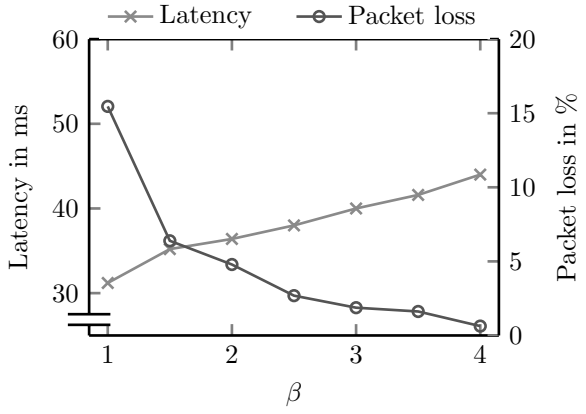


Figure 9: Latency and packet loss against packet loss tuning factor β

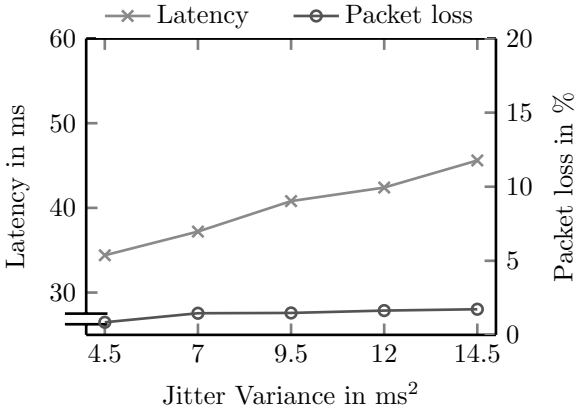


Figure 10: Adaption of latency and packet loss to various jitter conditions

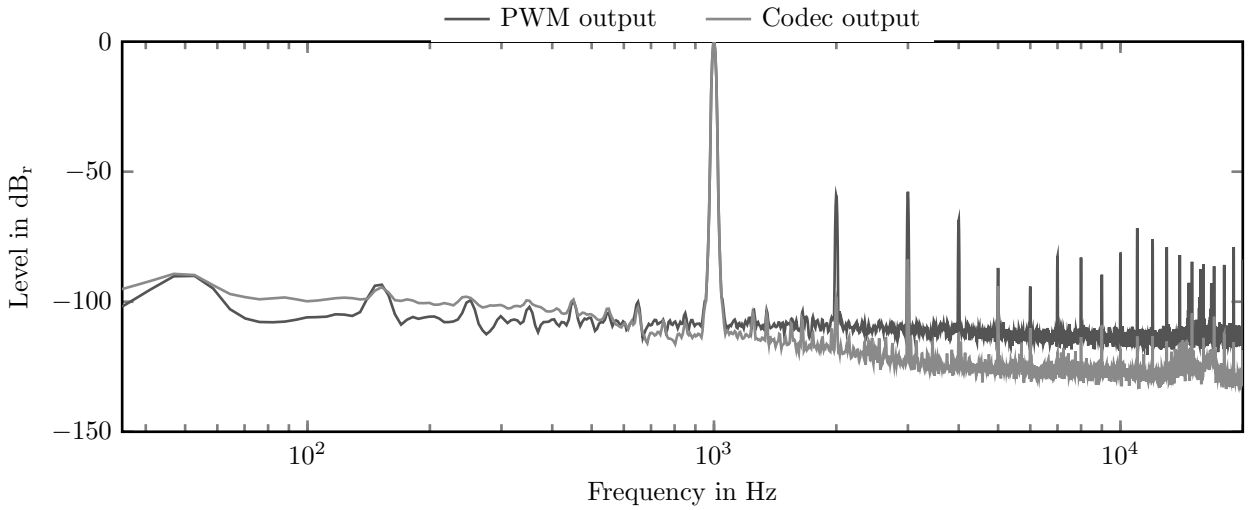


Figure 12: Spectra of the Raspberry Pi PWM and the codec board output for an 1 kHz sine

the THD of the new codec board is at least −20 dB lower than the original output for all analyzed signal levels.

These outcomes and the corresponding measurement results of the other audio hardware modules are listed in Tab. 1 and 2. The identified values should allow a high-quality capturing and playback of instruments. Analog amplification is always connected with the addition of noise. Therefore, the values of the input amplifier decrease with an increase of gain. For achieving even better quality, the flexibility of the device allows for connection of almost any kind of music equipment, like favored guitar amplifiers or vintage synthesizers.

7 Conclusions

The goal of this project was to create a standalone device, called the JamBerry, capable of delivering the well-known experience of a distributed network performance in a user-friendly way. The device is based on the famous Raspberry Pi and is enhanced by several custom hardware extensions: a digital and an analog extension board, providing high-quality audio interfaces to the Raspberry Pi, and a touchscreen to allow standalone operation of the device.

The performance was evaluated under lab conditions and the authors assume that the system and especially the audio quality shall satisfy the need of most musicians. Besides the described device design proposal, the main author shares the ALSA kernel driver that is included in the Linux mainline kernel since version 3.14 allowing the versatile connection of the Raspberry Pi with external audio hardware.

References

[1] The Raspberry Pi Foundation. Raspberry Pi Homepage. `www.raspberrypi.org`.

[2] Alexander Carôt and Christian Werner. Network Music Performance-Problems, Approaches and Perspectives. In *Proceedings of the "Music in the Global Village"-Conference*, Budapest, Hungary, September 2007.

[3] Alain Renaud, Alexander Carôt, and Pedro Rebelo. Networked Music Performance: State of the Art. In *Proceedings of the AES 30th International Conference*, March 2007.

[4] Chris Chafe, Scott Wilson, Al Leistikow, Dave Chisholm, and Gary Scavone. A Simplified Approach to High Quality Music and Sound over IP. In *Proceedings of the COST-G6 Conference on Digital Audio Effects (DAFx-00)*, Verona, Italy, December 2000.

[5] Juan-Pablo Cáceres and Chris Chafe. JackTrip: Under the hood of an engine for network audio. *Journal of New Music Research*, 39(3), 2010.

[6] Alexander Carôt, Torben Hohn, and Christian Werner. Netjack - Remote music collaboration with electronic sequencers on the Internet. In *Proceedings of the Linux Audio Conference (LAC 2009)*, Parma, Italy, April 2009.

[7] Alexander Cart and Christian Werner. Distributed Network Music Workshop with

Soundjack. In *Proceedings of the 25th Tonmeistertagung*, Leipzig, Germany, November 2008.

[8] Dimitri Konstantas, Yann Orlarey, Olivier Carbonel, and Simon Gibbs. The Distributed Musical Rehearsal Environment. *IEEE MultiMedia*, 6(3), 1999.

[9] Chrisoula Alexandraki and Demosthenes Akoumianakis. Exploring New Perspectives in Network Music Performance: The DIAMOUSES Framework. *Computer Music Journal*, 34(2), June 2010.

[10] European Broadcasting Union. Specification of the Digital Audio Interface (The AES/EBU interface), 2004.

[11] Stefan Schmitt and Jochen Cronemeyer. Audio over Ethernet: There are many solutions but which one is best for you? In *Embedded World*, Nürnberg, Germany, March 2011.

[12] Chris Chafe, Juan-Pablo Caceres, and Michael Gurevich. Effect of temporal separation on synchronization in rhythmic performance. *Perception*, 39(7), 2010.

[13] Alexander Carôt, Christian Werner, and Timo Fischinger. Towards a comprehensive cognitive analysis of delay influenced rhythmical interaction. In *Proceedings of the International Computer Music Conference (ICMC'09)*, Montreal, Quebec, Canada, August 2009.

[14] Advanced Linux Sound Architecture (ALSA) Project Homepage. `www.alsa-project.org`.

[15] Florian Meier. wamp_cpp - WAMP Server Implementation for C++. `github.com/koalo/wamp_cpp`.

[16] WAMP - the WebSocket application messaging protocol. `wamp.ws/spec`.

[17] Jean-Marc Valin, Koen Vos, and Timothy B. Terriberry. Definition of the Opus Audio Codec. Internet Engineering Task Force, RFC 6716, September 2012.

[18] Jean-Marc Valin, Timothy B. Terriberry, Christopher Montgomery, and Gregory Maxwell. A High-Quality Speech and Audio Codec With Less Than 10-ms Delay. *IEEE Transactions on Audio, Speech, and Language Processing*, 18(1), January 2010.

[19] Marco Fink, Martin Holters, and Udo Zölzer. Comparison of Various Predictors for Audio Extrapolation. In *Proceedings of the International Conference on Digital Audio Effects (DAFx'13)*, Maynooth, Ireland, September 2013.

[20] Colin Perkins, Orion Hodson, and Vicky Hardman. A Survey of Packet Loss Recovery Techniques for Streaming Audio. *IEEE Network*, 12(5), 1998.

[21] Benjamin W. Wah, Xiao Su, and Dong Lin. A Survey of Error-Concealment Schemes for Real-Time Audio and Video Transmissions over the Internet. In *Proceedings of the International Symposium on Multimedia Software Engineering*, Taipei, Taiwan, December 2000.

[22] I^2S Bus. Specification, Philips Semiconductors, June 1996.

[23] Alexander Carôt and Christian Werner. External Latency-Optimized Soundcard Synchronization for Applications in Wide-Area Networks. In *Proceedings of the AES Regional Convention*, Tokio, Japan, July 2009.

Case Study: Building an Out Of The Box Raspberry Pi Modular Synthesizer

Jürgen Reuter

Karlsruhe,
Germany,
reuter_j@web.de

Abstract

The idea is simple and obvious: Take some Raspberry Pi computing units, each as a reusable synthesizer module. Connect them via a network. Connect a notebook or PC to control and monitor them. Start playing on your virtual analog modular synthesizer. However, is existing Linux audio software sufficiently mature to implement this vision out of the box? We investigate how far we get in building such a synthesizer, what existing software to choose with focus on networking, analyse what limits we hit and what features still need to be implemented to make our vision become reality.

Keywords

Raspberry Pi, Virtual Anolog Modular Synthesizer, Distributed Networked Audio

1 The Vision

The popular *Raspberry Pi* (or, shortly, *RPi*)[Raspberry Pi Foundation, 2014b] is a small, cheap, yet powerful, computing unit with many I/O jacks with Linux/ARMv6 available as operating system (*OS*). It is predestinated for building networks of collaborative modules, with each RPi taking over the role of a synthesizer module with dedicated function as e.g. oscillator, filter or modulator. Using the RPi's *General Purpose I/O* (*GPIO*) pins or SPI interface, only minimal circuitry is required to equip the RPi with knobs (e.g. potentiometers or rotary encoders) or sliders, preferably on a separate, tiny board, also called a *shield*. This way, you get a distributed user interface, with knobs and sliders located directly on the module that it controls. Modules can be added to or removed from the network in a hot-plugging manner. If, for a different setup, you need, say, more oscillators and less modulators, you may change the role of a module simply by changing the software that it runs.

Compared with a virtual analog modular synthesizer running on a notebook or PC, the approach of a network of RPis reveals several advantages:

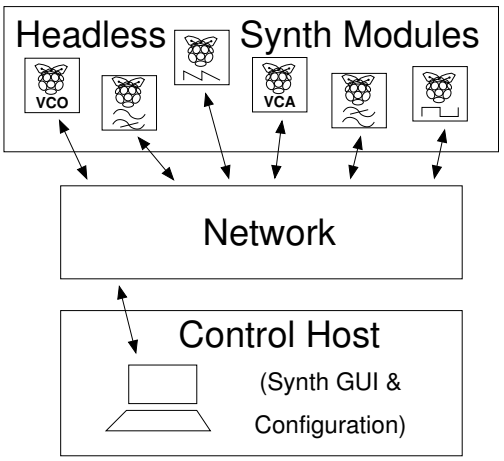


Figure 1: The Vision

- **Dedicated System.** The RPis are solely used for synthesis. The OS, residing on an SD card, can be tailored to this purpose. Many services are irrelevant for headless mode or use in a synthesizer and thus need not be installed, thus saving space and CPU time. The Linux audio RPi page[Linuxaudio.org, 2014] lists many tips for tuning overall latency. Once running, infrequent software updates should suffice, such that the chance to break the installed software e.g. with incompatible libraries can be reduced.
- **Distributed computing.** While the performance of the notebook or PC can easily become a bottleneck, in the distributed network audio computing performance scales with the number of modules.
- **Distributed interface.** With a single mouse and keyboard, you can control only a single input (such as a slider or knob) at one time. Optionally, the RPis can be

equipped with their own sliders and knobs, enabling to control them in parallel. Also, you may place the modules on, say, a large table, while on the virtual desktop of your notebook or PC, you are limited to size of your screen display.

- **Authenticity.** In a live performance, it is more comprehensible for the audience to see a musician put hands on physically existent modules and hear the resulting change of sound, rather than watching a PC user clicking and typing on his computer.

Still, a notebook or PC maybe useful as host for controlling network connections between the modules.

2 The Hardware

Our vision just integrates existing software and hardware, one may think. In fact, we have to carefully choose software that smoothely integrates with our RPis. We look at the RPi's hardware to better understand software requirements.

2.1 Audio Connections

For building an audio network, we have to decide what interconnects to use for audio transmission. Essential criteria are:

- **Duplex operation.** Synthesizer modules typically have both, input and output. We do not want to add hardware to gain full duplex operation.
- **Bandwidth.** Bandwidth must be sufficiently high for carrying multiple channels.
- **Audio and control data.** For low bandwidth data such as envelopes or frequency control, low bandwidth connections (e.g. MIDI) should be supported to save overall bandwidth.
- **Hardware protocol support.** To save computing resources, low-level issues (e.g. parity check bits or serializing / deserializing) should be implemented in hardware.

The RPi's hardware connectors capable of transmitting audio include the analog 3.5" audio output jack, USB, HDMI, GPIO / I2S, and Ethernet (Fig. 2).

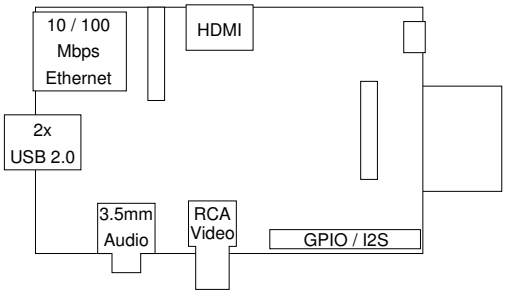


Figure 2: RPi Connectors Relevant for Audio

2.1.1 Analog 3.5" Audio Output Jack

Users report glitches, crackles and pops when using the 3.5" jack, at least in the early days of the RPi. It appearently supports only 11 bits of resolution [Linuxaudio.org, 2014]. Most important, the RPi has no analog audio input. Analog audio can not feed back into the RPi without additional hardware; hence we do not persue this jack.

2.1.2 USB

Linux supports audio over USB. The RPi model B's built-in USB 2.0 connectors only support connecting a USB device, but not another USB host[Raspberry Pi Foundation, 2012d]. Similar to the analog audio jack, symmetrical host-to-host transmission is impossible for RPi model B. In theory, RPi model A's single USB port can make the RPi act as device, via a host-to-host USB cable, but there does not seem anyone on the Web having confirmed that the USB drivers support this mode.

2.1.3 Audio via HDMI

While audio can be transmitted via HDMI, the RPi's built-in hardware does not support HDMI input; hence we do not persue this option.

2.1.4 GPIO pins

The RPi's GPIO pins are ideal for low-level input and output of binary data. For streaming audio data over GPIO, we would have to implement a full protocol stack in software, thus consuming much computing resources and limiting bandwidth. Specific GPIO pins implement I2S with hardware support[Raspberry Pi Foundation, 2012c]. While this interface may provide sufficient performance (users report varying experiences on this issue[stackexchange.com, 2013]), we would need special hardware (e.g. an I2S router). Some users claim that the kernel needs to be patched with an I2S kernel module to achieve high performance for audio data over I2S[GmbH, 2013]. At least, special I2S audio

drivers would have to be implemented to make audio applications aware of I2S.

2.1.5 TCP/IP or UDP over Ethernet

The RPi's Ethernet plug can be used for transmitting audio data. Neither TCP/IP nor UDP have been designed for realtime applications, but existing software for audio and video streaming over the internet shows that, with some effort, streaming is feasible as long as network bandwith is sufficiently high. We persue the approach of streaming audio over ethernet.

3 The Software

Preferring Ethernet for audio transmission, we next look into the software to choose and how to set it up. We prefer an out of the box solution of open source software.

3.1 Choosing Proper Audio Streaming Software

On Linux, there are competiting sound servers for streaming audio data over a network. In our short survey the following criteria are essential:

- **Availability.** The software must be available for ARMv6. Software, that is not open source or not under active development, is typically not available for this architecture.
- **Latency.** In modular synthesis audio and control data typically follow a path running through many modules. Therefore, the sound server should care for low latency.
- **Headless use.** The RPis will be typically controlled remotely and therefore run headlessly, that is without a display connected to them. No graphical environment such as X11 or a window manager should be required to run.

Audio streaming software like the *Enlightened Sound Daemon (ESD)* or *Phonon* are bound to a window manager. Therefore, ESD and Phonon are no valid candidates for our purpose. *sndio* is an audio server for OpenBSD, however, we are looking into a solution for Linux/ARM. The *aRts* sound server is out of development since 2006 and therefore not a viable choice. A quick internet search yields the following candidates:

- Network Audio System (NAS)
- PulseAudio
- JACK with netJACK

3.1.1 Network Audio System (NAS)

The Network Audio System (NAS)[radscan.com, 1996 2013] does not (yet) list any ARM architecture as supported platform. The man page of NAS states that the server automatically converts all data to the designed format or rate, that is, resampling may slow down overall performance.

3.1.2 PulseAudio

PulseAudio supports streaming over networks[freedesktop.org, 2013; archlinux.org, 2012 2014]. However, latency seems to be a major problem; only recently, major improvements have been announced[Lindner, 2013]. Also, PulseAudio resamples all data into some internal format and again resamples it for delivery. The RPi's limited computing performance should be saved for the actual audio processing of the synthesizer module's function.

3.1.3 JACK with netJACK

In contrast to PulseAudio and NAS[jack devel@jackaudio.org, 2012], JACK[jackaudio.org, 2006 2014] synchronizes all clients to one sound sink. JACK has been designed from the beginning with low latency in mind. Over the last few months, some remaining bugs that specifically appeared on the RPi/ARMv6 platform, have been fixed. We decide to pursue JACK, using (jackdmp) version 1.9.9. On our control host notebook, there was a pre-installed JACK (jackdmp) version 1.9.8 that we continue using. Any newer version should also work.

3.1.4 netJACK1 vs. netJACK2 vs. JackTrip

netJACK1 is available for both JACK1 and JACK2. In JACK1, netJACK1 is loaded with the command jackd -R -d net, while netJACK2 is not available. In JACK2, netJACK1 is loaded with jackd -R -d netone, while netJACK2 is loaded with jackd -R -d net.

The graphical application qjackctl can be used to load and configure netJACK1 or netJACK2 as backend. However, as of qjackctl version 0.3.9 bundled with current NOOBS, several bugs render qjackctl effectively useless when used with the netJACK2 backend on the RPi. Particularly, it uses netJACK1 options such as -o4 instead of -P 4 for setting up 4 output channels, and exhibited problems to detect an already running JACK instance. We prefer to use the RPis in headless mode anyway, i.e. without

using qjackctl. Running qjackctl on the control host seems to be fine for our purposes.

The documentation of JackTrip[Caceres and Chafe, 2010] explains how to setup a single JackTrip server with a single JackTrip client. The JackTrip server is a stand-alone application that, when started, appears as regular JACK client. When trying to start another JackTrip server instance, it complains that the associated UDP socket is already in use. The single JackTrip server instance spawns only a single readable client in qjackctl. The JackTrip documentation does not show any appearent way to connect multiple JackTrip clients. The latest ChangeLog entry of JackTrip dates from November 2010. As we need to connect multiple clients, we do not further persue JackTrip.

3.2 Putting it All Together

Next, we give a step by step instructions for installing and configuring all software for an RPi based modular synthesizer with JACK and netJACK.

3.2.1 Setting up NOOBS on the RPi

By now, there is no distribution tailored for audio applications on the RPi. Instead, we use the New Out Of Box Software (NOOBS) version 1.3.2 (Debian 3.10.24+ #614 PREEMPT armv6l kernel) based on the Wheezy Raspian distribution. We follow the instructions on the download webpage[Raspberry Pi Foundation, 2014a] and on the screen display, that we needed to attach to the RPi solely for the first installation, as well as a keyboard. Once one system is running, no more display or keyboard is needed. The SD card's contents can be copied to create another OS instance for another RPi.

When asked by NOOBS to select a distribution, we choose Raspian (i.e. Debian wheezy). With 16GB class 4 SD card and 10 MBit/s internet connection, the following installation roughly takes 45 minutes, including several automatic reboots. Finally, the raspberry configuration tool raspi-config is executed. The preset defaults should be fine.

JACK including netJACK should be already installed. For developing and compiling JACK clients, you need to install C header files for JACK with the command sudo apt-get install libjack-jackd2-dev, that installs the packages libdbus-1-dev and libjack-jackd2-dev. If there is no DHCP server on your network, do not forget to statically assign a unique IP address to each RPi and to set up a proper route to your network.

Now we have basic NOOBS installed on the RPi. Name for login on the RPi is pi, password is raspberry.

3.2.2 Setting Up JACK on the RPi

To automatically start a JACK slave instance on each RPi upon boot, put the following line into the /etc/rc.local script:

```
sudo -u pi /usr/bin/jackd -R -d net -n
module-name >/dev/null 2>&1 &
```

The instance will then appear to the master JACK instance on the control host as a remote slave instance called module-name. Alternatively, JACK can be started implicitly by the client application. Say, you have a low-pass filter implementation that connects to JACK with

```
jack_options_t options = JackNullOption;
client = jack_client_open (client_name,
         options, &status, server_name);
```

and your .jackdrc configuration file in your home directory containing the following line:

```
/usr/bin/jackd -R -d net -n low-pass \
               -C 3 -P 4
```

Then starting your client application will also start JACK. That is, in your /etc/rc.local script, you can also directly launch your client application.

3.2.3 Setting Up JACK on the Control Host

On our notebook we use JACK 1.9.8. The JACK master instance is started with the command jackd -R -d alsa or with whatever backend else you prefer over ALSA. After that, we load netJACK with the command jack_load netmanager, such that all JACK slaves on the RPis may connect to the JACK master.

Note that the JACK master on the control host must be started first. After that, boot the RPis. If JACK is automatically started on the RPi, then it will become visible to the control host. Use qjackctl on the control host to connect all RPis' inputs and outputs. Start playing your distributed RPi based modular synthesizer.

3.3 Synthesizer Software

Throughout this work, the author used very simple self-written JACK clients based on the simple_client.c example of the JACK distribution. In our out of the box spirit, we want to apply those existing LV^2 plugins[LV2, 2014]

for actual synthesis that do not require a GUI. Therefore we need a simple JACK client serving as host for LV^2 plugins that examines a given LV^2 plugin's I/O lines and exports them as JACK channels. The author does not know of an existing software doing this job, but the effort should not be too large. This work still has to be done.

4 Advanced Module Identity and Identification

For simplicity and ease of use, each RPi should represent exactly one synthesizer module. Then each RPi has a dedicated, clearly distinct function, helping to keep clear oversight over the whole system. This approach looks like waste of computing resources, if, for example, one RPi is dedicated as an oscillator. However, even tasks looking as simple as an oscillator may evolve into high complexity when adding sophisticated input controls, for example for morphing between sounds. We identify three options of identification: remote setup, SD card based identity, shield based identity.

4.1 Remote Setup

The software on the RPi's SD card contains all software for all supported module types, e.g. oscillator, low-pass filter or reverb effect. The function of a specific RPi is determined by remotely configuring it on the control host.

4.2 SD Card Based Identity

As the RPi uses its SD card as resident memory with complete OS and application software on it, the RPi gets a complete new identity by simply replacing the SD card. That is, we may create an oscillator SD card, a low-pass filter SD card, a reverb effect SD card, etc. The RPi represents a particular type of synthesizer module just by inserting the appropriate SD card. The RPi announces itself e.g. as oscillator or low-pass filter or reverb effect. The control host will collect all announcements and present all available modules to the user for wiring.

4.3 Shield Based Identity

For most modules, it is useful to provide hardware knobs or sliders directly attached to the modules, using a shield mounted directly on the RPi. While this approach requires (little) extra hardware and thus is not a pure out of the box solution, it has substantial advantages:

- **Visual module identification.** The extra hardware gives the RPi a visual identity and emphasizes that RPi's dedicated function. Each shield can be individually labelled (e.g. "master reverb effect") and typically has a set of knobs or sliders that also may help identify its function.
- **Parallel control of hardware knobs and sliders.** When controlling a virtual knob or slider on a screen, you need to place the mouse pointer on it. That is, only one input control can be used at a time. Switching between knobs or sliders takes time for relocating the mouse pointer. Hardware knobs and sliders enable parallel use and fast switching.
- **Module identity change by shield replacement.** If the shield provides an identifier for the RPi that represents the module's intended function (e.g. an LADSPA plugin ID), the RPi may automatically start any associated software or plug-in that implements the module's function indicated by the shield.

This way configuring an RPi as a dedicated module boils down to connecting a specific shield with knobs and sliders to it. The RPi's SD card holds the software for any supported module, and when plugging in a shield, the RPi can determine which module to represent.

4.4 Shield Design

While the author has not (yet) developed a shield, the design idea is simple and straightforward. We need circuitry connected to the RPi's GPIO pins that converts input from analog controls like knobs or sliders into digital signals. There are shields available with exactly this feature[abelectronics.co.uk, 2013; Raspberry Pi Foundation, 2012a; Modern Device, 2014]. Even cheap A/D converters are sufficient for low frequency signals such as movements of knobs and sliders[Sklar, 2012] and accessible via SPI[Brownell and others, 2013; Gzamboni, 2013]. Rotary encoders can be directly connected to the GPIO pins, as they simply consist of mechanical switches. The shield should contain a small serial EEPROM for uniquely identifying or describing the module's function and maybe storing a user-defined module label. The label must be stored on the shield, not on the RPi's SD card, as is names the module's function as stamped by the shield, regardless of the particular underlying RPi. Maybe designing and producing proper

shields for our synth modules can emerge as candidate for a tiny crowdfunding project.

5 Evaluation

Our attempt to set up a modular synthesizer using out of the box software shows remarkable limitations that should be considered as feature requests for the software that we discuss.

5.1 Audio Data Routing

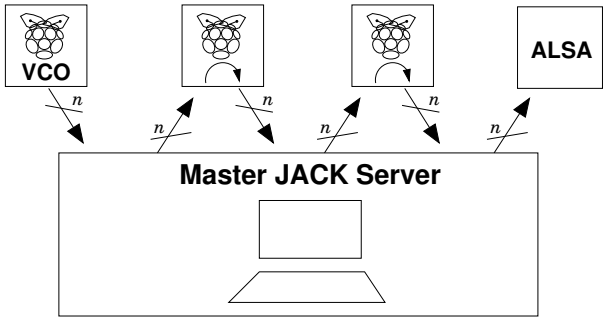


Figure 3: Routing via Master JACK Server

The probably most obstructive issue is the central routing of all audio data via the master JACK instance. The netJACK2 approach assumes a single master instance[Stéphane Letz et al., 2009]. Similarly, in netJACK1 a slave can only be connected to one master at a time. The master's backend (e.g. an ALSA sound device) determines the sample rate and format for all commnunication with all participating JACK slave instances. To prevent the master becoming a bottleneck, we would prefer RPi hosting a slave and a master JACK instance.

The qjackctl application follows the JACK and netJACK design and provides a GUI for configuring routing between the single master and multiple clients / slaves. If in a future version of JACK / netJACK multiple masters were supported, we would like to have an extended version of qjackctl capable of managing connections between two remote master instances.

For evaluating the impact of central routing, we measured the master JACK CPU load as a function of the number of audio channel connections. We connected three RPis to our notebook (Quad Core i5-2430M @ 2.4 GHz), used as control host for running the master JACK server. One of the RPis served as array of n oscillator outputs; the other two modules just looped through data from their n input channels to their n output channels. That is, for each channel audio data flows from the oscillator to the notebook, then to the first loop-through module and back to the notebook, then to the second loop-through module and back to the notebook and finally to the ALSA backend device (Fig. 3). That is, for n channels, there are $6n$ connections configured in qjackctl.

We measured the CPU load reported by jack_cpu_load() and varied the number of channels per module. Each box plot shows the range of CPU load of 480 samples (1sample/sec $\times$ 8min). For $n > 18$ (i.e. > 108 connections), severe problems like xrun errors and JACK crashes arised. Below this threshold, the system behaved smoothly with moderate load (Fig. 4). For comparison, the overall CPU load shown by xosview kept below 0.7.

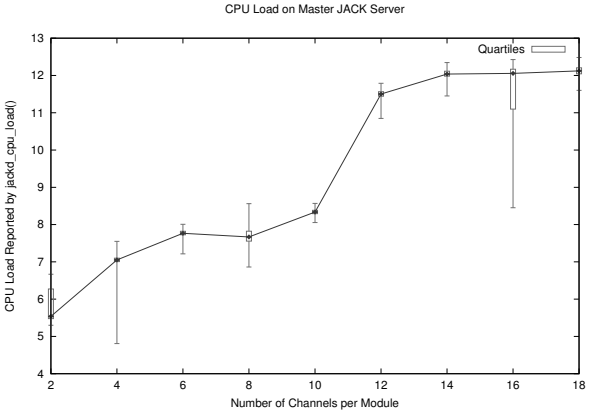


Figure 4: CPU Load on Master JACK Server

qjackctl often unexpectedly exited (with exit code 0), when a remote client reappeared after reboot. Sometimes it did not recognize when the connection to a remote client broke down due to client shutdown (maybe due to some problem of netJACK2). The master JACK server crashed when trying to connect more than 21 channels to the ALSA backend.

5.2 Labelling of Modules

On our control host notebook running qjackctl, by default all JACK slaves appear as clients with the name of the host they are running on. While this is reasonable default behaviour, we have a network of RPis with basically identical software setup. Therefore all RPis will appear as JACK clients labelled "raspberry" – the default host name for the Raspian Linux distribution. We could configure individual hostnames for each RPi, but it is the type of synthesizer module that should be displayed rather than a host identifier. Also, we do not want to change the host name each time the RPi changes the type of module. Luckily,

netJACK2 provides command line option -n to explicitly set the client name. For example, jackd -R -d net -n low-pass will result in a client called low-pass (Fig. 5). In netJACK1, there is no comparable option.

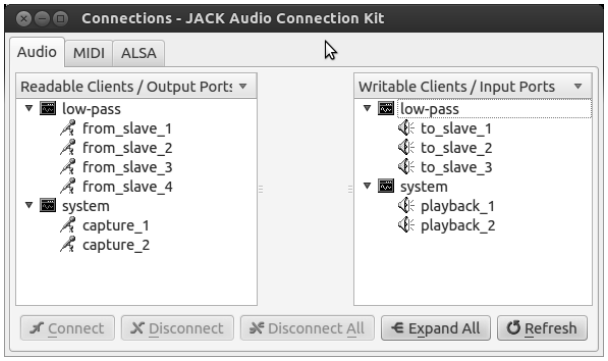


Figure 5: Slave started with jackd -R -d net -n low-pass -C 3 -P 4

5.3 Number of Channels

By default, a netJACK client is started with a number of audio input and output channels equal to that of the soundcard, i.e. mostly 2 for stereo sound. In contrast, a synthesizer module may have an arbitrary number of inputs and ouputs. Luckily again, the netJACK2 backend provides the command line options -C, -P, -i, and -o to specify the number of audio input, audio output, MIDI input, and MIDI output channels, respectively (Fig. 5). In netJACK1, the corresponding options are named -i, -o, -I, -O, respectively.

5.4 Labelling of Channels

The module software should be able to configure the names of a JACK slave's audio and MIDI input and output channels individually. For example, an oscillator may have a pitch control input, a noise content control input, a left channel audio output and a right channel audio output.

In JACK, port names can be set with the function jack_port_set_name(jack_port_t *port, const char *port_name). They are initially set to capture_n or playback_n for inputs or outputs, respectively. When the function is executed on the RPi, it refers to the port name locally shown on the RPi. Unfortunately, netJACK does not report slave port names to the master. Instead, on the master JACK instance, remote client channels always appear as from_slave_n, to_slave_n, midi_from_slave_n, and midi_to_slave_n (Fig. 5).

Of course, if software running on the control host has knowledge about all synthesizer module types that may appear in the network, it may derive labels channels labels from the slave's name. This workaround however does not qualify as out of the box solution.

5.5 Boot Time

Our NOOBS based setup takes almost one minute of time for booting an RPi. For an embedded system that you want to immediately start working with, this time is far too long. There exist tailored kernels and software configurations for faster booting, reduced to a minimum of what is required for the dedicated purpose. Choosing a fast SD card may speed up booting as well as using the kernel's fastboot option. RPi users report tips und tricks to reduce its boot time to as low as less than 20 seconds[Raspberry Pi Foundation, 2012b].

6 Conclusions

Linux on Raspberry Pi is *almost* mature for implementing our vision of a modular synthesizer based on a network of RPis connected to a notebook or PC as control host. The most outstanding problem in our setup is that *all* audio and MIDI data is routed through the master JACK instance running on the control host. Instead, the RPis should be able to communicate directly with each other. This approach however would require multiple JACK master instances to be part of the communication network, while the netJACK architecture currently assumes a single master instance.

As the RPis are run in headless mode, on the control host we would like to run an application capable of configuring the complete system. In particular, setting up direct connections between two RPis (once it will be supported) reaches beyond the scope of qjackctl. An extended version of qjackctl could turn out essential for our vision. The overall stability of qjackctl should be improved.

For using existing LV^2 plugins, there is missing some JACK client serving as LV^2 plugin host.

The author plans to get in contact with the authors of the depicted software to solve remaining issues. The example & testing code of this study is available at http://www.soundpaint.org/rpi-modular-synth/. The author wants to thank the anonymous reviewer for pointing out some missing point for true out of the box spirit and a further reference.

References

abelectronics.co.uk. 2013. ADC Pi - 8 Channel Analogue to Digital converter for the Raspberry Pi computer boards ADC PIV2. http://www.abelectronics.co.uk/products/3/Raspberry-Pi/17/ADC-Pi-V2---Raspberry-Pi-Analogue-to-\Digital-converter.

archlinux.org. 2012–2014. Pulseaudio/examples - archwiki. https://wiki.archlinux.org/index.php/PulseAudio/Examples#PulseAudio_over_network.

David Brownell et al. 2013. spidev. https://www.kernel.org/doc/Documentation/spi/spidev.

Juan-Pablo Caceres and Chris Chafe. 2010. JackTrip: JackTrip Documentation. https://ccrma.stanford.edu/groups/soundwire/software/jacktrip/.

freedesktop.org. 2013. Network. http://www.freedesktop.org/wiki/Software/PulseAudio/Documentation/User/Network/.

Modul 9 GmbH. 2013. Hifiberry dac – linux configuration — crazy audio. http://www.crazy-audio.com/projects/hifiberry-mini/hifiberry-mini-linux-configuration/.

Gzamboni. 2013. SPIdev. http://linux-sunxi.org/SPIdev.

jack devel@jackaudio.org. 2012. Discussion of the jack audio server and jack applications: Netjack for Thinclients Instead of Pulseaudio. http://comments.gmane.org/gmane.comp.audio.jackit/25406.

jackaudio.org. 2006-2014. JACK — connecting a world of audio. http://jackaudio.org/.

Mirko Lindner. 2013. PulseAudio 4.0 verringert Latenz und steigert Geschwindigkeit - Pro-Linux. http://www.pro-linux.de/news/1/19858/pulseaudio-40-verringert-latenz-und-\steigert-geschwindigkeit.html.

Linuxaudio.org. 2014. Raspberry Pi and realtime, low-latency audio [Linux-Sound]. http://wiki.linuxaudio.org/wiki/raspberrypi#on-board_audio.

LV2. 2014. LV2 Trac. http://lv2plug.in.

Modern Device. 2014. Lots of Pots Board for Raspberry Pi. http://moderndevice.com/product/lots-of-pots-lop-board-for-raspberry-pi.

radscan.com. 1996–2013. The Network Audio System (NAS). http://www.radscan.com/nas.html.

Raspberry Pi Foundation. 2012a. Gertboard — Raspberry Pi. http://www.raspberrypi.org/archives/tag/gertboard.

Raspberry Pi Foundation. 2012b. How to speed up boot time if run headless? http://www.raspberrypi.org/phpBB3/viewtopic.php?f=29&t=25777.

Raspberry Pi Foundation. 2012c. Raspberry Pi – I2S: Anyone got it running? (answer is yes!). http://www.raspberrypi.org/phpBB3/viewtopic.php?t=8496.

Raspberry Pi Foundation. 2012d. Raspberry Pi - Model A Q: Can it be USB client instead of USB host? http://www.raspberrypi.org/phpBB3/viewtopic.php?f=63&t=15696.

Raspberry Pi Foundation. 2014a. Downloads — Raspberry Pi. http://www.raspberrypi.org/downloads.

Raspberry Pi Foundation. 2014b. Raspberry Pi — An ARM GNU/Linux box for $25. Take a byte! http://www.raspberrypi.org.

Mikey Sklar. 2012. Analog Inputs for Raspberry Pi Using the MCP3008 — Adafruit Learning System. http://learn.adafruit.com/reading-a-analog-in-\and-controlling-\audio-volume-with-the-raspberry-pi/overview.

stackexchange.com. 2013. hardware – How fast is GPIO+DMA? Multi I2S input – Raspberry Pi Stack Exchange. http://raspberrypi.stackexchange.com/questions/9646/how-fast-is-gpiodma-multi-i2s-input.

Stéphane Letz et al. 2009. Walk-Through/User/NetJack2 Jack Audio Connection Kit – Trac. http://trac.jackaudio.org/wiki/WalkThrough/User/NetJack2.

Experimenting with a Generalized Rhythmic Density Function for Live Coding

Renick BELL
Tama Art University
2-1723 Yarimizu
Hachioji, Tokyo 192-0394
Japan
renick@gmail.com

Abstract

A previously implemented realtime algorithmic composition system with live coding interface had rhythm functions which produced stylistically limited output and lacked flexibility. Through a cleaner separation between the generation of base rhythmic figures and the generation of variations at various rhythmic densities, flexibility was gained. These functions were generalized to make a greater variety of output possible. As examples, L-systems were implemented, as well as the use of ratios for generating variations at different rhythmic densities. This increased flexibility should enable the use of various standard algorithimic composition techniques and the development of new ones.

Keywords

algorithmic composition, live coding, Haskell, L-systems, rhythm

1 Introduction

A system for realtime algorithmic composition was first presented in (Bell, 2011) and then improvements were described in (Bell, 2013). The intention of that system was to be able to do realtime algorithmic composition, primarily through a live coding interface. It was concluded that while the interonset interval (IOI) function and density function provided in the Conductive library could yield somewhat useful results, improvements could be made.

This paper describes attempted improvements in this area. First this paper briefly reviews how those functions were implemented previously and describes their output. It then explains the problems with that implementation and output. The paper then proceeds to describe the newly implemented version and its advantages, namely the use of a higher-order function to gain a more modular system. Two example inputs to this higher-order function were implemented and are described. Finally, conclusions are made and directions for future research are proposed.

2 Summary of Previous Rhythm Generation Technique

2.1 A Brief Review of Conductive

Conductive is a library for the Haskell programming language used for managing concurrent processes for realtime music. In addition to providing functions for managing those concurrent processes, it has some features for representing musical time and for algorithmic composition.

Concurrent musical processes are represented by a data structure called a Player. The Player refers to two functions: an action, which can be any IO function, which it runs repeatedly; and an IOI function, which determines the wait times, called interonset intervals (IOIs), that are interleaved between calls to the action function. More information on Conductive and performing with it can be found in (Bell, 2011) and (Bell, 2013). See Figure 1 for a graphical representation, originally included in (Bell, 2011).

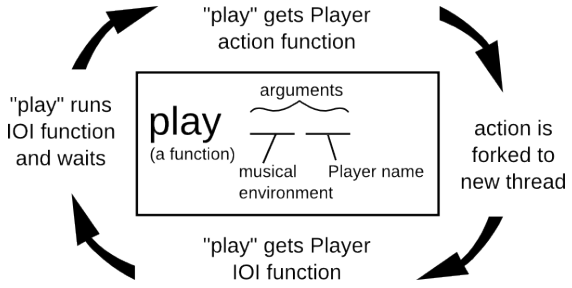


Figure 1: the Play loop, with Player, action function, and IOI function

2.2 Rhythm in Conductive

As described in (Bell, 2013), this author had been experimenting with reading IOI values from something called a density map. This concept involved two parts: the generation of rhythmic figures and the generation of an ordered stack of variations at lesser and greater rhythmic density. Both of these parts used

stochastic methods and were joined finally in a single function.

A higher-level abstraction was developed to generate a list and a set of rhythmically similar lists of greater and lesser density and store them in a table indexed by level of density. Doing so increased the likelihood that two lists would be perceived as having a rhythmic relationship and decreased the chance that an audience would perceive a kind of discontinuous state change when switching from one to another.

The base rhythmic figure was an ordered list of IOI values expressed in terms of beats, whole or fractional. To generate it, a performer selected a core unit which was used to generate potential IOI values. Selection of the core unit, in conjunction with the length of the pattern, largely determined the metrical feel of the pattern. A list of scalars was determined by the performer, from which the function randomly selected a user-specified quantity to multiply with the core unit. The user specified a number of subphrases to generate and the length of those phrases in terms of number of scalars to use. The subphrases were then generated by selecting the scalars and multiplying them by the base unit. Finally, a user-specified number of subphrases were chosen at random by the function. The user determined the length of the final phrase in terms of beats. If the length of the concatenated subphrases did not equal the specified length, the final IOI value was padded. If the length exceeded the specified length, the final IOI value was truncated. The repetition of values and subphrases within the final figure tended to give it a musical quality lacking in a list of purely random numbers. For a complete example, see (Bell, 2013).

Given a particular figure, a series of related patterns was generated in which the rhythmic density was increased or decreased. A large number of such patterns was generated in order so that a stack of patterns from very low rhythmic density to very high rhythmic density resulted, with the original figure somewhere in the middle. Those variations were generated in one of two ways, depending on whether they were to have greater or lesser density. When reducing density, one value from the figure was chosen at random and combined with a neighboring value. That new value was inserted into the figure in the place of those two selected values. The less dense variation was then subjected to the same process recursively until the figure contained only a single item, with the figure at each step added to a list. For increasing density, a value was chosen at random and replaced with two items: an item of lesser value from a list of potential IOI values and the difference between the original IOI value and the lesser value. The resulting figure was subjected to the same process, again recursively and retaining each version, until the figure consisted of a list of the smallest of the potential IOIs. By concatenating the original figure with the lists of greater and lesser density, a table was generated. For a complete example of this, see (Bell, 2013).

An integer value representing the density ranking, with 0 being the least dense pattern, was assigned to each figure in the table. That table, along with the list of potential IOIs and the total length of the figure, was stored in a data structure called an IOIMap.

The function call to execute this process looks like this, containing Ints, Doubles, and lists of each as arguments. The function returned an IOIMap containing the density map based on the generated rhythmic figure:

```
m00 <- iOIandRTfromPhrase 0.25 2
          2 4 [2] 2 [2] 4.0 3
```

Based on a user-specified density value, a particular IOI pattern is chosen from the table. The user queries the table with a value between 0 and 1, and a linear conversion to a list index is done. The value returned is the IOI pattern at that index. Based on the current beat, an IOI value is returned from that pattern.

Density values can vary with time. One method for doing so is employing a TimespanMap. A commonly observed pattern was setting the timing of particular values. It is often desired that values change over time but at different rates. TimespanMaps are structures for handling such cases. Rather than specify the exact timing of a value, it specifies the range of time in which that value can occur. They are maps or dictionaries with intervals as keys to any kind of value. Another parameter of the structure is a specified length at which it loops. When a time is passed to the dictionary, the interval that time falls in is determined to be the key to use, and the corresponding value for that interval is returned. When the time value passed to the TimespanMap exceeds those for which it is defined, it loops to return an appropriate value. For a graphical explanation, see Figure 2 in (Bell, 2013).

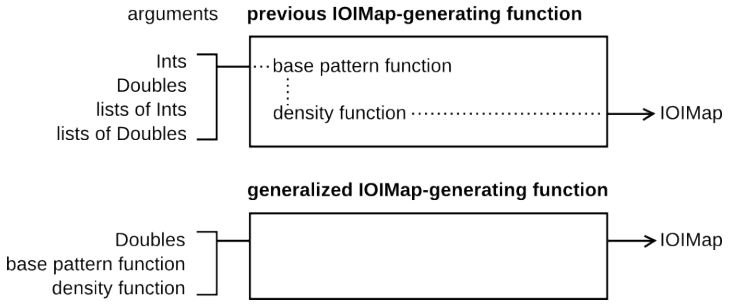


Figure 2: a comparison of the previous method and the new method

2.3 Problems Identified in the Previous Paper

The style of patterns produced was limited, both in the generation of rhythmic figures and their variations at various rhythmic densities. More inconvenient was the fact that other methods could not be tested without rewriting the core functions. A more modular solution was sought.

3 New Method: a Generalized Function for Patterns and Variations

To solve the problem described above, it was determined to rewrite the function that handled density and base patterns, generalizing it to take functions and make the density function a higher-order function. A higher order function is "a function that takes a function as an argument or returns a function as a result is called higher-order." (Hutton, 2010) In this case it takes two functions as parameters: one for the generation of the base pattern and a second for determining how the density of an input should be increased. The function is passed those two functions and a parameter determining how long the rhythmic figures should be. It uses the pattern function to produce a base pattern. It then processes the base pattern with the density function to create the rhythmic variations. It returns an IOIMap, which includes a set of TimespanMaps mapping time intervals to values of next beats ordered according to their density value.The new function has been given the temporary name of newIOIMap2, to be used until a better method of naming it is determined.

The benefit of doing so is that the basic structure is already available before a performance and does not need to be coded at that time or recoded when the current method for generating patterns is longer useful. That means that generating patterns and then making a table of values which can be read from according to a density value can be accomplished more easily and in a greater variety of ways. John Hughes writes in his essay "Why Functional Programming Matters" that higher-order functions are one of two important kinds of "glue" that increase modularity, "the key to successful programming". (Hughes, 1989) The use of higher-order functions in programming for aesthetic output has been described in (McDermott et al., 2010).

As an example of an alternate method for increasing density, a function for increasing density by ratio has been implemented. As an example of a function for generating base rhythms, two types of L-systems were implemented.

3.1 Density by Ratio

This method applies when generating variations of increased rhythmic density.

The user specifies a list of ratios, a lowest target value, and a limit to how small the IOI values in the rhythmic figure can be. A value is selected at random from the rhythmic figure. A ratio is chosen at random from the list of ratios provided by the user. The ratio is applied to the value, which is subtracted from the original value. These two new values are then shuffled and inserted into the rhythmic figure in place of the original value. The new rhythmic figure is stored in a list, and the process is repeated on this figure. This process is carried out recursively until all of the IOI values in the rhythmic figure are equal to or less than the user-specified lowest target value, producing a stack of increasingly dense rhythmic figures. The code for this procedure can be seen in the functions "densifier" and "densifier2".

Consider an example in which a list, [1,1,1,1] is progressively densified according to a ratio of 0.5, with 0.25 being the lowest possible value.

```
let addL unit phraseLength ratios name = do
        ioimap <- newIOIMap2 0 phraseLength (generateDensities2 unit ratios) $ lsysTest4
        rs +@ (name,ioimap)
```

Figure 3: an example of using the new higher order function, newIOIMap2

In this example, "it" is the ghci reference to the output of the previous command.

```
*> densifier 0.25 [0.5] [1,1,1,1]
[1.0,1.0,1.0,0.5,0.5]
*> densifier 0.25 [0.5] it
[0.5,0.5,1.0,1.0,0.5,0.5]
*> densifier 0.25 [0.5] it
[0.5,0.5,1.0,0.5,0.5,0.5,0.5]
*> densifier 0.25 [0.5] it
[0.5,0.5,0.5,0.5,0.5,0.5,0.5,0.5]
*> densifier 0.25 [0.5] it
[0.25,0.25,0.5,0.5,0.5,0.5,0.5,0.5,0.5]
```

This process would continue until all of the values in the list are equal to 0.25.

3.2 Explanation of L-systems

Lindemayer systems, abbreviated to L-systems, were developed by Aristid Lindenmayer in 1968 as "a theory of growth models for filamentous organisms" (Lindenmayer, 1968). They have since been used by many for the simulation of plant growth, for visual art, and to a lesser extent, music.

They are string-rewriting systems in which an input string, called an axiom, is transformed according to a set of rules in which each item in the string (a predecessor symbol) is rewritten as a successor string. By inputting this output back through the rule-set, successive generations can be obtained (DuBois, 2003).

Here is a small example of a rule set, input, and seven generations (Supper, 2001):

```
rules: a -> b
       b -> ab

input: a

output: b
        ab
        bab
        abbab
        bababbab
        abbabbababbab
        bababbababbabbababbab
```

L-systems which do not have one-to-one string replacement rules grow in length rapidly as seen above and require users to employ techniques to deal with that size (DuBois, 2003).

3.3 History of L-systems in Algorithmic Composition

L-systems have been used in a variety of ways for algorithmic composition. Some examples from the literature are listed below.

L-systems are frequently used for pitch content. Supper describes the use of L-systems and cellular automata for algorithmic composition (Supper, 2001). Langston used L-systems to choose from previously composed musical phrases (Langston, 1989). Morgan uses a system somewhat similar to Langston in which previously generated pattern fragments are chosen from and assembled according to an L-system-generated template (Morgan, 2007). One of the most complete and useful discussions of using L-systems for music is the dissertation by R. Luke Dubois, which describes various methods for generating patterns of pitches in monophonic melody lines and chords.

Worth and Stepney describe a set of L-system-selected rules by which note duration is progressively transformed (Worth and Stepney, 2005). Use of L-systems for duration or rhythm are described by Kaliakatos-Papakostas, Floros, et. al., including the use of what they call FL-systems, in which L-system output is constrained in length (Kaliakatsos-Papakostas et al., 2012). Kitani and Koike have also described a method of generating rhythms from L-systems in combination with a learning algorithm (Kitani and Koike, 2010). Liou, Wu, and Lee use L-systems to compute the complexity of rhythms (Liou et al., 2010).

For more about L-systems, readers are referred to the dissertations of Dubois (DuBois, 2003) and Manousakis (Manousakis, 2006) first and then the other items listed above.

3.4 The L-system Function Implemented for this System

The initial intention for using L-systems with the higher-order function described above is to generate the base rhythms from which the density table described above can be generated.

The module itself contains functions for generating a string output from an axiom, a rule

set, and the generation number. Using the map function, a set of several generations can be obtained.

The rule set is notated with a colon rather than the traditional arrow for speed of entry. The predecessor symbol and successor string are written without spaces and separated by a colon. Each production rule must be separated by a space. The previous example can be run in ghci as follows:

```
*LSystem> let rules = "a:b b:ab"
*LSystem> getGeneration2 1 rules "a"
"a"
*LSystem> getGeneration2 2 rules "a"
"b"
*LSystem> getGeneration2 3 rules "a"
"ab"
*LSystem> getGeneration2 4 rules "a"
"bab"
*LSystem> getGeneration2 5 rules "a"
"abbab"
```

A more complicated example follows:

```
rules: "a:ab b:acd d:gx e:abc f:ga g:d"
axiom: "abcdefg"
```

A symbol which has no rule is kept as-is. In is the same as if the rule were to repeat the symbol, such as "c -> c".

In this case, the output in the interpreter of the first three generations of this L-system are:

```
*LSystem> getGeneration2 1 "a:ab b:acd
d:gx e:abc f:ga g:d" "abcdefg"
"abcdefg"
*LSystem> getGeneration2 2 "a:ab b:acd
d:gx e:abc f:ga g:d" "abcdefg"
"abacdcgxabcgad"
*LSystem> getGeneration2 3 "a:ab b:acd
d:gx e:abc f:ga g:d" "abcdefg"
"abacdabcgxcdxabacdcdabgx"
```

Two methods have been tested:

- direct output of IOI values
- lists of value-transforming functions applied in sequence to a base value

The direct output of IOI values means that given an axiom, a rule set, the generation number, and a list of potential IOI values, the function will return a list of IOI values. How those IOI values are assigned to the symbols is a matter for which a large variety of options exist. One simple choice is to randomly assign a value to each unique symbol. The string is then rewritten as that list of numeric values. This example illustrates such a method. The list of values ranges from 0.25 to 1.25, containing every step of 0.25.

```
*LSystem> getGeneration2 5 rules "a"
"abbab"
*LSystem> let a = it
*LSystem> randomFinalizer2
[0.25,0.5..1.25] a
[1.25,0.25,0.25,1.25,0.25]
*LSystem> randomFinalizer2
[0.25,0.5..1.25] a
[1.25,1.0,1.0,1.25,1.0]
*LSystem> randomFinalizer2
[0.25,0.5..1.25] a
[0.5,0.75,0.75,0.5,0.75]
```

In the case of using transform, the rules of the L-system are mathematical functions that modify a numerical value. First the output of the L-system is similarly rewritten with one mathematical function randomly chosen for each unique symbol in the string. Given a starting value and that list of mathematical functions, the number is passed through the list so that the output of one function becomes the input of the next. The changes are accumulated so that each step in the transformation of the initial number is kept. That series of numbers is then processed as deltas on which the density function will generate variations. Here is a very simple example of a list of functions processing a value:

```
*> transform 2 [(2 + ),((-1) +),(3 *)]
[2,4,3,9]
```

Here is an application of using the transform function on the output of an L-system.

```
*LSystem> getGeneration2 5 "a:b b:ab"
"a"
"abbab"
*LSystem> let a = it
*LSystem> let b = nub a
*LSystem> b
"ab"
*LSystem> let c = zip (map (\x -> [x])
b) [(1+),(0.5*)]
*LSystem> let d = flatFinalizer c a
*LSystem> :t d
```

```
d :: [Double -> Double]
*LSystem> transform 2 d
[2.0,3.0,1.5,0.75,1.75,0.875]
```

In this example, the variable "d" is the output of the function flatFinalizer, which converts the symbols in a string to their equivalents in a dictionary. In this case, the dictionary is "c", which maps the output of the L-system, "b", to the list of operations above. The ghci command ":t" shows the type of something, and in this case is used to show that d is a list of functions which take a Double and return a Double. The nub function returns a list in which all duplicate items have been removed. In this case, "d" would expand to:

```
*LSystem> transform 2 [(1+),(0.5*),
(0.5*),(1+),(0.5*)]
[2.0,3.0,1.5,0.75,1.75,0.875]
```

The final output in both of these examples, i.e. the list of Doubles, is used as a list of IOI values. Those values are then processed into a density map as described in sections 2 and 3.1.

4 Conclusion

An example of the function call now used to generate the IOIMap is shown below, including the partially applied function *generateDensities2* and the function *lsysTest4* can be found in Figure 3.

A brief example of using the L-system-based method described above can be heard at this URL: http://renickbell.net/sound/renick-bell-fractal-beats-test-140209-b.mp3

In this example, a collection of 100 density maps was created from a single L-System – "a:ab b:c c:abc d:ded e:aabb f:ga g:d" "abcdefg" – using the direct random selection of IOI values described above. Through the performance, 17 of those are auditioned using a variety of audio sample sets as well as live modification of the envelopes which control the density level.

Another brief example can be found at: http://renickbell.net/sound/renick-bell-fractal-beats-140125.mp3

In the near future, this code should be cleaned and added to one of the Conductive packages at Hackage, the Haskell package repository. A rough version of the code can be found in the meantime at this URL: http://renickbell.net/code/generalized-density.zip

With the modifications described above, the system certainly gained an additional degree of freedom. The system should now serve better as a platform for testing various algorithmic composition techniques. That could include more complex L-systems, stochastic systems, and so on.

Using ratios for increasing density works fairly well as long as the ratios are very simple, like 0.5. Other ratios generate patterns that are likely less familiar to listeners, and thus might not be appropriate if the composer has the intention of producing music that neatly fits within most existing genres. However, this was just an example, and more sophisticated methods can now be more easily tested.

The use of L-systems is also interesting, but it will take additional practice to become acquainted with L-systems and how, in the middle of a performance, to write rules and axioms for interesting output. As with the density function above, these L-system functions are also simple examples that can be refined or replaced for future work. They simply demonstrate that generalization was possible.

5 Acknowledgements

I would like to thank Akihiro Kubota and Yoshiharu Hamada for research support.

6 References

Renick Bell. 2011. An Interface for Realtime Music Using Interpreted Haskell. In *Proceedings of LAC 2011*. editions.

Renick Bell. 2013. An Approach to Live Algorithmic Composition using Conductive. In *Proceedings of LAC 2013*. editions.

Roger Luke DuBois. 2003. *Applications of generative string-substitution systems in computer music*. Ph.D. thesis, Columbia University.

John Hughes. 1989. Why functional programming matters. *The Computer Journal*, 32(2):98–107.

Graham Hutton. 2010. *Programming in Haskell*. Univ. Press, Cambridge [u.a., editions.

Maximos A. Kaliakatsos-Papakostas, Andreas Floros, Nikolaos Kanellopoulos, and Michael N. Vrahatis. 2012. Genetic evolution of L and FL-systems for the production of rhythmic sequences. In *Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference*, pages 461–468. ACM, editions.

Kris M. Kitani and Hideki Koike. 2010. Improvgenerator: Online grammatical induction for on-the-fly improvisation accompaniment. In *Proceedings of the 2010 Conference on New Interfaces for Musical Expression (NIME 2010)*. editions.

Peter Langston. 1989. Six techniques for algorithmic music composition. In *15th International Computer Music Conference (ICMC), Columbus, Ohio, November*, pages 2–5. Citeseer, editions.

Aristid Lindenmayer. 1968. Mathematical models for cellular interactions in development I. Filaments with one-sided inputs. *Journal of theoretical biology*, 18(3):280–299.

Cheng-Yuan Liou, Tai-Hei Wu, and Chia-Ying Lee. 2010. Modeling complexity in musical rhythm. *Complexity*, 15(4):19–30.

Stelios Manousakis. 2006. Musical L-systems. *Koninklijk Conservatorium, The Hague (master thesis)*.

James McDermott, Jonathan Byrne, John Mark Swafford, Michael O'Neill, and Anthony Brabazon. 2010. Higher-order functions in aesthetic EC encodings. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8. IEEE, editions.

Nigel Morgan. 2007. Transformation and mapping of L-Systems data in the composition of a large-scale instrumental work. In *Proceedings of ECAL 2007 Workshop on Music and Artificial Life (MusicAL 2007)*. editions.

Martin Supper. 2001. A few remarks on algorithmic composition. *Computer Music Journal*, 25(1):48–53.

Peter Worth and Susan Stepney. 2005. Growing music: musical interpretations of L-systems. In *Applications of Evolutionary Computing*, pages 545–550. Springer, editions.

Live-Coding-DJing with Mixxx and SuperCollider

Antonio José Homsi GOULART
Computer Music Research Group
Instituto de Matemática e Estatística - Universidade de São Paulo
Rua do Matão, 1010 - Cidade Universitária - 05508-090 - São Paulo, SP - Brazil
ag@ime.usp.br

Abstract

This paper suggests a modified dance music DJ performance, based on common DJing techniques enriched with live coding moments either mixed with records or not, instead of only reproducing previous-made tracks. That way, all the different possibilities offered by live coding are put together with commercial tracks, promoting live coding while maintaining the dance music atmosphere and opening more improvisation possibilities for the DJ. It is also a funny way to start learning programming and live coding. All software involved are open-source and the workflow is based on the author's. The primary intention here is to stimulate DJs to try live coding, at the same time helping to promote its sonority to audiences other than experimental music enthusiasts.

Keywords

live coding, DJing, live performance, education

1 Introduction

A DJ work consists of researching a lot to find the most suitable tracks for a specific venue and selecting a good order to play them as a set. Also important is the ability to mix, which consists of beat-matching 2 tracks and make a transition from one to another, maintaining a continuous flow instead of separately starting each track [Broughton and Brewster, 2006] [Collins et al., 2013]. Mixxx[1] [Andersen, 2003] is an open-source digital DJing software suited for that.

Another type of live music performance is live coding, also called on-the-fly programming, in which the programmer/performer augments or modifies code while it is running and generating real-time sound, without the need to stop or restart the program [Wang and Cook, 2004]. A lot of languages are suited for this task, and one of the most popular is the open-source SuperCollider[2] [McCartney, 2002].

[1] http://www.mixxx.org/

[2] http://supercollider.sourceforge.net/

In this paper a workflow based on a mixture of DJing and live coding will be described. I believe that documenting this process, which is a very simple one but presents some technicalities, might benefit or stimulate seasoned DJs to incorporate new tools to their sets. At the same time I hope that it helps promoting live coding at mainstream venues and open-source software to more users.

In sections 2 and 3 DJing and live coding practices will be detailed. Section 4 describes the live-coding-DJing method. Conclusions and perspectives are analyzed in section 5.

2 DJing

The term DJ comes from Disk Jockey, referring to the act of playing vinyl records for an audience. In a typical configuration, a DJ would work with two turntables and a mixer, so two songs could be reproduced simultaneously, an important issue to make the transitions.

When CDs became available more and more people switched to the smaller and cheaper discs and CD Decks, and nowadays many DJs work only with a laptop computer. "(...) If they lugged large boxes of records with them in the 1990s, after the millennium a tendency was evident toward lighter-weight luggage allowed by fully digital track management in hard-drive disk jockeying" [Collins et al., 2013].

No matter which media is chosen the DJ task is the same: playing records one after the other, mixing them, which consists of beat-matching (getting the tracks to play at the same tempo, with their beats synchronized [Broughton and Brewster, 2003]) and then making the transition, which can be a blend (a gradual fade out of the first track and fade in of the second) or a cut (a sudden change of track being played).

The beat-matching stage is typically done with a headphone: while the current song is reproduced for the audience, the next one is reproduced on the DJ's headphones. Then the

tempo might be adjusted and after that the beats synchronized. Instead of using the headphones, one could beat-match by looking at the tracks' waveforms, assisted by tempo and beat detection software. There is also the more extreme possibility of doing it by ear dispensing headphones aid, mixing while both songs are reproduced to the audience.

Some techniques [Broughton and Brewster, 2003] might make the blend more interesting:

- a simple blend using the faders: fade one track out and the other one in (fade durations depend on the genres);
- matching phrases: dance music tracks are divided in 4-bar phrases. Usually there are clues at the end of these phrases, e.g. a cymbal crash, extra drum beats or an instrument finishing a solo. When mixing, it is important to match phrases and beats;
- take advantage of keys, and avoid key clashes: some tracks sound bad when mixed because their notes are not in the same key. Some alternatives in this situation are mixing when one (or both) is just percussion or pitch-shift one of them;
- equalization can be used to hide parts of a song while keeping others: one example is removing the bass line from a track and introducing the bass line of a new song, then do the same with the other parts;
- matching rhythms: some tracks fit together better than others, because besides their harmonies and melodies match, their rhythms fit perfectly together. Difficulties arise when mixing two tracks with syncopation, or too many drumbeats.

Instead of blending a track with the subsequent one a DJ may cut, which is switching sharply from one record to another without losing the beat. Cuts tend to sound better with sparse, percussive music, and bad with tracks containing continuous melodies [Broughton and Brewster, 2003]. Other alternatives are doing stops (stop current track and then start the next record after a while) or spin-backs (reverse current track and then cut to the next one).

Some techniques were discussed but we shall keep in mind that, as emphasized by Broughton and Brewster, the most important thing about DJing is choosing the records and the order to play them, and after that the crucial decision is where to put the joins. "Where the mix occurs is more important than how it occurs" [Broughton and Brewster, 2003].

A DJ software (or any task-specific application) ties the user to its paradigms. All actions, control structures, and interaction possibilities are defined in advance. With a rigid interface, they offer high visibility of available operations and immediate gestural control for the live performer to adapt sound immediately and continuously, but with reduced potential for creative exploration [Blackwell and Collins, 2005].

Mixxx's interface (Figure 1)[3], for example, contains buttons for loading tracks in 2 different virtual decks, playing, stopping and looping them, setting cue points (important points in the track, likely to be replayed), knobs and faders for adjusting effects parameters, filters, tempo and playback rate for each track, among other functionalities, including a waveform visualizer for the loaded tracks.

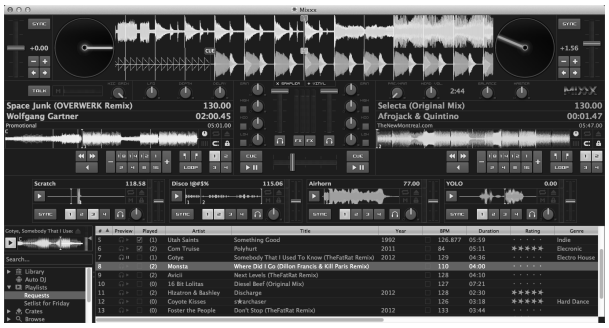


Figure 1: Mixxx interface (Late Night Blues).

3 Live-coding

On the other hand comes the possibility of working with interpreted computer languages, modifying running algorithms on-the-fly and, in the case of audio programming, getting real-time sound as the program's output. Brown [Brown, 2006] defines live coding as a practice where "digital content is created through computer programming as a performance".

The interface for live coding is only a text editor[4] and a shell for feedback (although fancy IDEs are available), which means that all the actions are hidden in text commands. All the exploratory possibilities of computer languages are available for the performer, but no inter-

[3] picture from http://www.mixxx.org/press/

[4] For text-based languages. Graphical programming languages, where programming is made linking objects in a canvas are also available, e.g. PureData.

face with buttons prepared for interaction are present. Therefore code must be written for all the actions involved in sound production, which can cause a high mental load specially in the scarce-time situation of a performance.

Blackwell and Collins argument that besides aesthetic reasons a key concern for someone to choose the challenge of live coding instead of using a pre-made software is that these "are biased towards fixed audio products in established stylistic modes, rather than experimental algorithmic music which requires the exploratory design possibilities of full programming languages" [Blackwell and Collins, 2005].

Different approaches can be taken in a live coding session, from a low level and mathematical one, writing complex algorithms that output sound as they evolve, to a more high level and simple approach, describing synthesis models and then sequencing sounds by specifying values for the models' parameters. These values can result from routines evaluation or be directly chosen by the user. When working with this instrument plus sequencer paradigm, code can be viewed as a description of instruments and a score (a scheduling of musical events).

This last approach represents in my opinion the easiest way to live code, provided the artist can describe a synthesis model, even if a simple one. Programming notes for melodies/harmonies and rhythm patterns is simple, so it can perfectly be combined with dance music. This might be a naive approach for live coding, but it is enough to synthesize sounds within the LCDJ paradigm to be proposed.

3.1 The performance

In live coding performances usually the code is projected for the audience. According to Brown, most people like it but some people find it a display of virtuosity and distraction to the listening experience [Brown, 2006]. Alex McLean, who plays raves and programmers gatherings said "I prefer it when the audience is dancing and doesn't care how we're making the music" [Andrews, 2006].

A live coding session may start either from scratch or with previous written code. In this last case, the artist may augment, modify or fill blanks. As time is scarce in a performance an interesting trick is to prepare snippets of code with the basic structure and syntax of the language (or even snippets with the main sounds and patterns of a personal production).

An alternative approach for live coding is doing it in duos or larger groups, or even in an orchestra together with musicians playing acoustic instruments. That takes the pressure off a single performer and, in the case of a live coding group, shares the algorithmic complexity between the members [Collins, 2011].

4 Why not doing both? LCDJ!

If someone is neither restricted to perform like a DJ nor like a live coder, and if someone can access tools for both occupations within the same computer, why not playing DJ styles of music adding some live coding moments, or why not live code supported by nice tracks playing along? Why not doing it if both tools are running on the same sound server?

In an interview, dance music duo Coldcut said "The future of DJing is not about whether vinyl will survive. The future of DJing is about media mixing. The DJ with two SL1200s [turntables] will fade out, but if he is clever, he'll evolve into a multi-armed posse manipulating various sound and vision sources. There should be a new name for this, maybe a media-jockey" [Broughton and Brewster, 2003].

4.1 Suggestions for LCDJing

A LCDJ may start the performance playing a record or coding (releasing initial sounds after a while). In the second case the mood can be set according to improvisational decisions made exactly at the time of the performance, so a sound that perfectly invokes the intended atmosphere can be synthesized, instead of having to pick one from the finite set that is the hard drive.

A LCDJ is able to jam with records via live code, inventing new sounds or mimicking/emphasizing/satirizing the record's. Stopping the track for a solo is also a good move. In the case of playing a personal production, different versions of it can be improvised by modifying code used to generate it (and that's a good appeal to produce music using code); that can also be a way to tease the audience revealing pieces of the upcoming track, an effect similar to cutting back and forth between two beat-matched tracks [Broughton and Brewster, 2003].

Another interesting option is routing audio from Mixxx to SuperCollider, processing tracks with infinite possibilities of effects, instead of only applying some high-pass or low-pass filters or common effects like flanging or ring modulation (these are the options available in Mixxx's and other similar programs interfaces).

In my experience, mixing is where LCDJ true potential is revealed. Instead of blending or cutting like a DJ, a live-coding-DJ may live code between tracks. A simple guide for that can be:

1. choose the next song to play in the set and load it in Mixxx;
2. choose an instant in the current song to start interacting;
3. start live-coding and interact (in any way) with the current track;
4. when current track ends, take the live-coding to a sonority suited to welcome the subsequent track;
5. choose a moment and start the new track;
6. when it is suited, stop live coding and let the new track fly.

Step 4 can be made at any pace and some ways to welcome a track are by:

- invoking its rhythm;
- invoking its bass line, melody or harmony;
- making a sparse and percussive sound, preparing for a cut;
- making a totally non-sense sonority that brings tension to be released with next track (perfect for tracks with a sweet and melodic intro).

Of course there is always the option of not welcoming a track and live code for hours.

The blending techniques mentioned earlier can be adapted for LCDJing. Some suggestions:

- matching phrases: mimic a bass line or melody of current song and keep playing it for a while until the song vanishes. Then adapt it to an element present in next track, start it and interact for a while;
- keys: start coding with current track, at the same key. When it ends, progressively add notes from another key, but without clashing. When the sonority has been taken to the same key as the next track, all is set for a good entrance;
- equalization: a great chance to modify tracks. Cut some parts of the current track, for example, the bass line, and live code a new one. When suited, introduce a sonority that resembles next track and call it;
- match rhythms: be the drummer, along the current track, then solo, then with the new one. Link tracks using percussive lines.

The techniques presented are the ones I could come up with and test in some occasions, but there is no limit for the possibilities in LCDJing, besides what one can do with code. Feeling the audience, the mood and the venue style are important clues for how far from mainstream a LCDJ can go in a session.

4.2 Software involved

There are lots of nice applications for DJing[5] and languages suited for live coding[6]. Depending on personal choices any combination of software can be used for LCDJing. One could even dispense DJ software and LCDJ using only a language. In my experience Mixxx and SuperCollider is a good pair for Live-Coding-DJing performances because:

- both are very efficient, so LCDJing is possible even with a NetBook;
- Mixxx is very easy to learn and provides all the tools a DJ need[7], so common DJ actions can be readily done instead of having to code them;
- although a first contact with SuperCollider might be frightening, its syntax makes it easy and fast to code synth models and sequence patterns (all we need to LCDJ);
- both connect to Jack[8], which allows audio routing between software, so Mixxx and SuperCollider can communicate, sending and receiving audio to and from each other. Some advantages and possibilities have already been discussed;
- both are open-source, with all the related advantages.

4.3 Simple Mixxx, to collide with SuperCollider

Both newcomers and artists used to other DJ applications will find it intuitive to work with Mixxx. Its interface is really simple and everything necessary for LCDJ is available as a shortcut in the computer keyboard. A quick

[5] http://linux-sound.org/ddj.html
[6] http://toplap.org/wiki/ToplapSystems
[7] http://www.mixxx.org/features/
[8] http://jackaudio.org/

read in the wiki[9] and one is familiarized. The community forums[10] are also good resources.

In the author's opinion a good practice for DJing, specially LCDJing, is to avoid mouse (time is scarce) and external controllers (save money and space in the cabin/backpack and make use of the laptop hardware as a whole).

4.4 Simple SuperCollider, to mix with Mixxx

As it was highlighted above, the live coding strategy proposed here is very simple; only instrument definitions and a way to sequence sounds are necessary. For the instruments definitions the class *SynthDef* is used. Its syntax is shown in Figure 2, with a simple sawtooth oscillator and an ADSR envelope being defined.

```
SynthDef(\saw_ex, {
    |out=0, gate=1, pan=0, freq=100,
     a=0.1, d=0.2, s=0.8, r=0.3,//env args
     mix=0.5, room=0.8| //reverb arguments
    var env = EnvGen.kr(Env.adsr(a,d,s,r),
                  doneAction:2, gate:gate);
    var snd = Saw.ar(freq);
    snd = snd * env;
    snd = Pan2.ar(snd, pan);
    snd = FreeVerb.ar(snd, mix, room);
    Out.ar(out, snd);
}).add;
```

Figure 2: Defining an instrument.

The classes *Pdef* and *Pbind* might be used for the sequencing of sounds. The syntax is presented in Figure 3, with a pattern that repeats the notes D,F,A (the degrees 1,3,5 are converted intro *freq* values) sequentially, along with values for each note duration and panning. *Pseq* picks the argument vector values in a sequence.

Notice that each note attack time and reverb mix (dry=0/wet=1) will have a random value (*Prand* randomly picks values from the argument vector), so the instrument sound will always be changing. Models with more parameters can offer a wide timbre variation.

The equivalent of this 30-line simple and flexible implementation would hardly (if even possible) be attained in more rigid interfaces. Other options for sequencing and more complex synthesis models (and much more information) can be found in the learning SuperCollider page[11] and SC community[12].

[9] http://www.mixxx.org/wiki/doku.php
[10] http://www.mixxx.org/forums/index.php
[11] http://supercollider.sourceforge.net/learning/
[12] http://supercollider.sourceforge.net/community/

```
Pdef(\play_saw,
    Pbind(
      \instrument, \saw_ex,
      \degree, Pseq( [ 1,3,5 ] , inf),
      \dur, Pseq( [ 1,1,2 ] , inf),
      \a, Prand(0.1*[5,11,22,33], inf),
      \d, 0.5, \s, 0.95, \r, 0.15,
      \pan, Pseq([-1,1,-0.5,0.5], inf),
      \mix, Prand([0.2,0.5,0.9], inf),
      \room, 0.5
    )
).play;
```

Figure 3: Specifying parameters values and sequencing sounds.

5 Conclusions

DJing is a long-time established practice and live coding a not so old one but it certainly has already established its importance in contemporary music practice. The workflow described in this paper does not intend to dismiss such practices, only mix them in a way that is simple for the seasoned DJ or anyone to try live coding and benefit. At the same time it is a funny and stimulating way to start programming, dive into synthesis studies and learn more about open-source software. Surely it only opens new possibilities for the artist.

Describing synthesis models, although a difficulty task at first, is definitely worth. The sound palette of the producer will grow to the point that besides having personal productions/tracks, a characteristic sound can also be achieved. In a world where most commercial electronic dance music sound so alike producing tracks and timbres is a good way for promotion. Sharing snippets of code with nice sounds and/or patterns also seems to be a good idea.

A pure live coding session aiming experimental music requires much more than only these simple concepts presented here. With this approach, however, a good level of interaction with dance music is possible because of its structure, which is usually rhythmic and well defined, with distinctive melodies and harmonies.

Whatever the genre of electronic music the DJ wants to play, interaction with live coding is possible - from abstract Ambient sounds to the rhythmic beats of mainstream House - even with the simple paradigm described in this paper. Synthesis models can be as varied as the creativity/ability of the artist; the instruments can be sequenced with a fixed or (widely) vary-

ing timbre; the rhythm and notes patterns can also be freely specified, even randomly (Why not going for a track that wasn't intended, just because a random pattern resembled it?).

Although studio productions are not the intended output of a LCDJ session, extra care must be taken with the rawness of the synthesized audio. Mainstream dance music records are equalized, pre-mixed, well-balanced, compressed and mastered, so in order to fit in new sounds some sculpting is necessary, otherwise they get the foreground and mask the record. Usually, extra equalizing in the record plus a little reverb and good positioning with panning (that's why they are in the example) in the live coding sounds are enough to find them a spot and prevent clashes.

A true improvising door opens with LCDJ. Although DJs know specific tracks to invoke different types of emotions, and DJing is based on improvisation according to the audience mood, the set of possibilities is finite, unless music is created on-the-fly.

The same way that a performance in group relieves part of the pressure on each artist, live coding along records also has the same effect. More time is available to analyze and shape sounds, impose a rhythm and write code.

Screen projection, although explored in pure live coding sessions, may be discarded in LCDJ. Code may be too simple, it would be a distraction for dancers and a spoiler for the set. However it depends on the venue, as more advanced programmers and specific audiences might like.

Of course the practice is not restricted to Mixxx and SuperCollider. Great software and languages are available for DJing (xwax, terminatorX, etc.) and live coding (ChucK, PureData, etc.). However, Mixxx's interface might be more familiar for seasoned DJs, especially those who work with turntables/decks or OSs other than Linux, and SuperCollider efficiency, along with Patterns Library - easy to learn and use - makes it a good option to start.

LCDJing would also be possible dispensing the DJ software and using only a live coding language. However, a DJ application facilitates performing common DJ tasks (creation/management of a playlist in the performance, adjusting tempo with a knob twist, cueing points in tracks and scratching), relieving the mental load that coding every move would create.

My impression on playing as a LCDJ is that people accept rhythmic live coding moments as unknown yet good track passages, when appropriately presented and not overdone. More abstract coding moments brings tension and curiosity, which calls that magical record.

6 Acknowledgements

I would like to thank CAPES for financial support and Flávio Schiavoni for suggestions on this work, and I wish that those who haven't tried live coding yet feel stimulated to do it, specially children learning music, music producers and DJs. Starting LCDJ in duos is also great!

References

Tue Haste Andersen. 2003. Mixxx: Towards novel dj interfaces. In *Proceedings of the 2003 Conference on New Interfaces for Musical Expression*, NIME '03, pages 30–35, Singapore, Singapore. National University of Singapore.

Robert Andrews. 2006. Real djs code live. *Wired, technology news.* Retrieved 01/20/2014 from ≪ http://www.wired.com/science/discoveries/news/2006/07/71248 ≫.

Alan Blackwell and Nick Collins. 2005. The programming language as a musical instrument. In *Proceedings of Psychology of Programming Interest Group*, pages 120–130.

Frank Broughton and Bill Brewster. 2003. *How to DJ right: The art and science of playing records.* Grove Press, New York.

Frank Broughton and Bill Brewster. 2006. *Last night a DJ saved my life.* Headline Book Publishing, London.

Andrew R. Brown. 2006. Code jamming. *M/C: a journal of media and culture*, 9(6), December. Retrieved 01/19/2014 from ≪ http://journal.media-culture.org.au/0612/03-brown.php ≫.

N. Collins, M. Schedel, and S. Wilson. 2013. *Electronic Music.* Cambridge Introductions to Music. Cambridge University Press.

Nick Collins. 2011. Live Coding of Consequence. *Leonardo*, 44(3):207–211.

James McCartney. 2002. Rethinking the computer music language: Supercollider. *Comput. Music J.*, 26(4):61–68, December.

Ge Wang and Perry R. Cook. 2004. On-the-fly programming: Using code as an expressive musical instrument. In *Proccedings of the International Conference on New Interfaces For Musical Expression*, pages 138–143.

Audio Signal Visualisation and Measurement

Robin Gareus
Université Paris 8
Paris, France
robin@gareus.org

Chris Goddard
Freelance audio engineer
Woodstock, Oxfordshire, UK
chris@oofus.co.uk

Abstract

The authors offer an introductory walk-through of professional audio signal measurement and visualisation.

The presentation focuses on the SiSco.lv2 (Simple Audio Signal Oscilloscope) and the Meters.lv2 (Audio Level Meters) LV2 plugins, which have been developed since August 2013. The plugin bundle is a super-set, built upon existing tools with added novel GUIs (e.g ebur128, jmeters,..), and features new meter-types and visualisations unprecedented on GNU/Linux (e.g. true-peak, phase-wheel,..). Various meter-types are demonstrated and the motivation for using them explained.

The accompanying documentation provides an overview of instrumentation tools and measurement standards in general, emphasising the requirement to provide a reliable and standardised way to measure signals.

The talk is aimed at developers who validate DSP during development, as well as sound-engineers who mix and master according to commercial constraints.

Keywords

Audio Level Metering, Visualisation, LV2, DSP

1 Introduction

Audio level meters are very powerful tools that are useful in every part of the production chain:

- When tracking, meters are used to ensure that input signals do not overload and maintain reasonable headroom.
- Meters offer a quick visual indication of activity when working with a large number of tracks.
- During mixing, meters provide a rough estimate of the loudness of each track.
- At the mastering stage, meters are used to check compliance with upstream level and loudness standards, and to optimise the dynamic range for a given medium.

Similarly for technical engineers, reliable measurement tools are indispensable for the quality assurance of audio-effects or any professional audio-equipment.

2 Meter Types and Standards

For historical and commercial reasons various measurement standards exist. They fall into three basic categories:

- Focus on **medium**: highlight digital number, or analogue level constraints.
- Focus on **message**: provide a general indication of loudness as perceived by humans.
- Focus on **interoperability**: strict specification for broadcast.

For in-depth information about metering standards, their history and practical use, please see [Brixen, 2010] and [Watkinson, 2000].

2.1 Digital peak-meters

A Digital Peak Meter (DPM) displays the absolute maximum signal of the raw samples in the PCM signal (for a given time). It is commonly used when tracking to make sure the recorded audio never clips. To that end, DPMs are calibrated to 0dBFS (Decibels relative to Full Scale), or the maximum level that can be represented digitally in a given system. This value has no musical connection whatsoever and depends only on the properties of the signal chain or target medium. There are conventions for fall-off-time and peak-hold, but no exact specifications. Furthermore, DPMs operate on raw digital sample data which does not take inter-sample peaks into account, see section 2.7.

2.2 RMS meters

An RMS (Root Mean Square) type meter is an averaging meter that looks at the energy in the signal. It provides a general indication of loudness as perceived by humans.

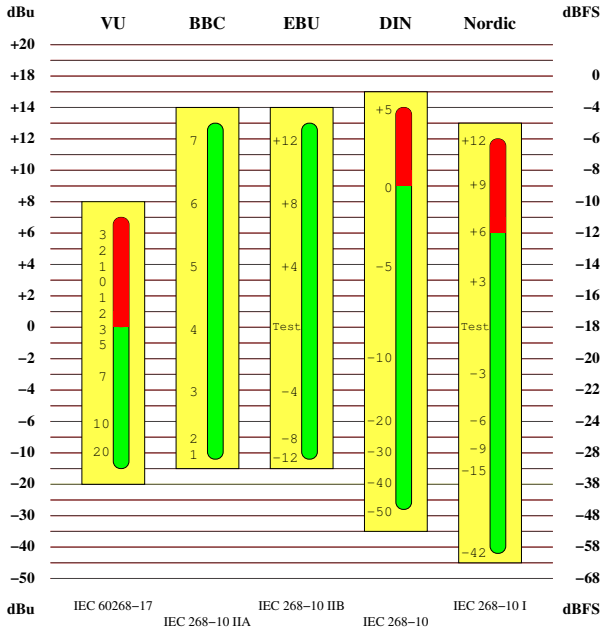


Figure 1: Various meter alignment levels as specified by the IEC. Common reference level is 0dBu calibrated to -18dBFS for all types except for DIN, which aligns +9dBu to -9dBFS. dBu refers to voltage in an analogue system while dBFS to digital signal full-scale.

Bar-graph RMS meters often include an additional DPM indicator for practical reasons. The latter shows medium specifics and gives an indication of the crest-factor (peak-to-average power ratio) when compared to the RMS meter.

Similar to DPM's, there is is no fixed standard regarding ballistics and alignment level for a general RMS meter, but various conventions do exist, most notably the K-system introduced by Bob Katz [Katz, 2000].

2.3 IEC PPMs

IEC (International Electrotechnical Commission) type Peak Programme Meters (PPM) are a mix between DPMs and RMS meters, created mainly for the purpose of interoperability. Many national and institutional varieties exist: European Broadcasting Union (EBU), British Broadcasting Corporation (BBC), Deutsche Industrie-Norm (DIN),.. [Wikipedia, 2013].

These loudness and metering standards provide a common point of reference which is used by broadcasters in particular so that the interchange of material is uniform across their sphere of influence, regardless of the equipment used to play it back. See Fig. 1 for an overview of reference levels.

For home recording, there is no real need for this level of interoperability, and these meters are only strictly required when working in or with the broadcast industry. However, IEC-type meters have certain characteristics (rise-time, ballistics) that make them useful outside the context of broadcast.

Their specification is very exact [IEC, 1991], and consequently, there are no customisable parameters.

Figure 2: Various meter-types from the meter.lv2 plugin bundle fed with a -18 dBFS 1 kHz sine wave. Note, bottom right depicts the stereo phase correlation meter of a mono signal.

2.4 EBU R-128

The European Broadcast Union recommendation 128 is a rather new standard, that goes beyond the audio-levelling paradigm of PPMs.

It is based on the ITU-R BS.1770 loudness algorithm [ITU, 2006] which defines a weighting filter amongst other details to deal with multi-channel loudness measurements. To differentiate it from level measurement the ITU and EBU introduced a new term 'LU' (Loudness Unit) equivalent to one Decibel[1]. The term 'LUFS' is then used to indicate Loudness Unit relative to full scale.

In addition to the average loudness of a programme the EBU recommends that the 'Loudness Range' and 'Maximum True Peak Level' be measured and used for the normalisation of audio signals [EBU, 2010].

[1] the ITU specs uses 'LKFS', Loudness using the K-Filter, with respect to to Full Scale, which is exactly identical to 'LUFS'.

The target level for audio is defined as -23 LUFS and the maximum permitted true-peak level of a programme during production shall be -1 dBTP.

The integrated loudness measurement is intended to quantify the average program loudness over an extended period of time, usually a complete song or an entire spoken-word feature. [Adriaensen, 2011], [EBU, 2011].

Many implementations go beyond displaying range and include a history and histogram of the Loudness Range in the visual readout. This addition comes at no extra cost because the algorithm to calculate the range mandates keeping track of a signal's history to some extent.

Three types of response should be provided by a loudness meter conforming to R-128:

- **Momentary** response. The mean squared level over a window of 400ms.
- **Short** term response. The average over 3 seconds.
- **Integrated** response. An average over an extended period.

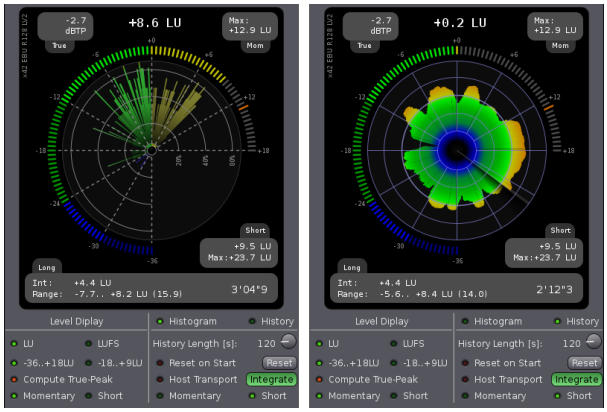


Figure 3: EBU R-128 meter GUI with histogram (left) and history (right) view.

2.5 VU meters

Volume Unit (VU) meters are the dinosaurs (1939) amongst meters.

The VU-meter (intentionally) "slows" measurement, averaging out peaks and troughs of short duration, and reflects more the perceived loudness of the material [Wikipedia, 2014], and as such was intended to help program producers create consistent loudness amongst broadcast program elements.

In contrast to all the previously mentioned types, VU metes use a linear scale (in 1939 logarithmic amplifiers were physically large). The meter's designers assumed that a recording medium with at least 10 dB headroom over 0 VU would be used and the ballistics were designed to "look good" with the spoken word.

Their specification is very strict (300ms rise-time, 1 - 1.5% overshoot, flat frequency response), but various national conventions exist for the 0VU alignment reference level. The most commonly used was standardised in 1942 in ASA C16-5-1942: "The reading shall be 0 VU for an AC voltage equal to 1.228 Volts RMS across a 600 Ohm resistance"[2]

2.6 Phase Meters

A phase-meter shows the amount of phase difference in a pair of correlated signals. It allows the sound technician to adjust for optimal stereo and to diagnose mistakes such as an inverted signal. Furthermore it provides an indication of mono-compatibility, and possible phase-cancellation that takes place when a stereo-signal is mixed down to mono.

2.6.1 Stereo Phase Correlation Meters

Stereo Phase Correlation Meters are usually needle style meters, showing the phase from 0 to 180 degrees. There is no distinction between 90 and 270 degree phase-shifts since they produce the same amount of phase cancellation. The 0 point is sometimes labelled "+1", and the 180 degree out-of-phase point "-1".

2.6.2 Goniometer

A Goniometer plots the signal on a two-dimensional area so that the correlation between the two audio channels becomes visually apparent (example in Fig. 8). The principle is also known as Lissajous curves or X-Y mode in oscilloscopes. The goniometer proves useful because it provides very dense information in an analogue and surprisingly intuitive form: From the display, one can get a good feel for the audio levels for each channel, the amount of stereo and its compatibility as a mono signal, even to some degree what frequencies are contained in the signal. Experts may even be able to determine the probable arrangement of microphones when the signal was recorded.

2.6.3 Phase/Frequency Wheel

The Phase Wheel is an extrapolation of the Phase Meter. It displays the full 360 degree signal phase and separates the signal phase by

[2]This corresponds to +4dBu

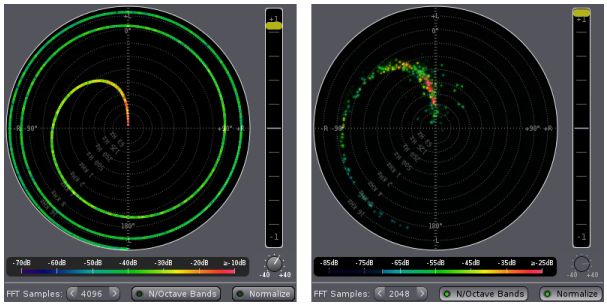


Figure 4: Phase/Frequency Wheel. Left: pink noise, 48KSPS with right-channel delayed by 5 samples relative to left channel. Right: Digitalisation of a mono 1/2" tape reel with slight head misalignment.

frequency. It is a rather technical tool useful, for example, for aligning tape heads, see Fig. 4

2.7 Digital True-Peak Meters

A True-Peak Meter is a digital peak meter with additional data pre-processing. The audio-signal is up-sampled (usually by a factor of four [ITU, 2006]) to take inter-sample peaks into account. Even though the DPM uses an identical scale, true-peak meters use the unit dBTP (decibels relative to full scale, measured as a true-peak value – instead of dBFS). dBTP is identical to dBFS except that it may be larger than zero (full-scale) to indicate peaks.

Inter-sample peaks are not a problem while remaining in the digital domain, they can however introduce clipping artefacts or distortion once the signal is converted back to an analogue signal.

floating point audio data	mathematical true peak value
.. 0 0 +1 +1 0 0 ..	+2.0982 dBTP
.. 0 0 +1 -1 0 0 ..	+0.7655 dBTP

Table 1: True Peak calculations @ 44.1 KSPS, both examples correspond to 0dBFS.

Fig. 5 illustrates the issue. Inter-sample peaks are one of the important factors that necessitate the existence and usage of headroom in the various standards, Table 1 provides a few examples of where traditional meters will fail to detect clipping of the analogue signal.

2.8 Spectrum Analysers

Spectrum analysers measure the magnitude of an input signal versus frequency. By analysing the spectra of electrical signals, dominant frequency, power, distortion, harmonics, bandwidth, and other spectral components can be observed. These are not easily detectable in time domain waveforms.

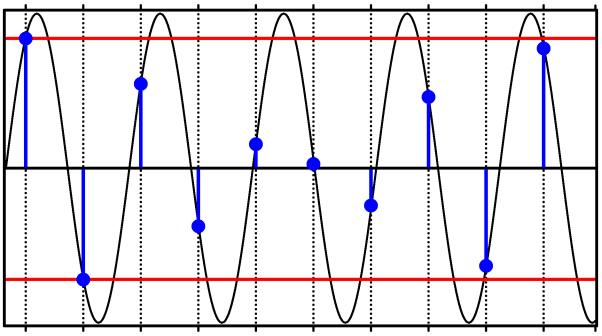


Figure 5: Inter-sample peaks in a sine-wave. The red line (top and bottom) indicates the digital peak, the actual analogue sine-wave (black) corresponding to the sampled data (blue dot) exceeds this level.

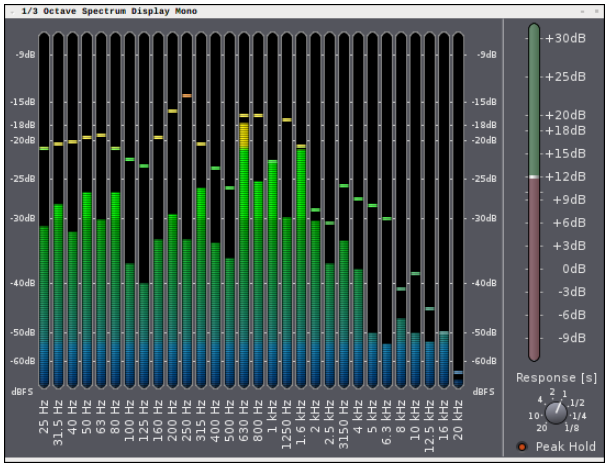


Figure 6: 30 Band 1/3 octave spectrum analyser

Traditionally they are a combination of band-pass filters and an RMS signal level meter per band which measures the signal-power for a discrete frequency band of the spectrum. This is a simple form of a perceptual meter. A well known specification is a 1/3 octave 30-band spectrum analyser standardised in IEC 61260 [IEC, 1995]. Frequency bands are spaced by octave which provides a flat readout for a pink-noise power spectrum, which is not unlike the human ear.

As with all IEC standards the specifications are very precise, yet within IEC61260 a number of variants are available to trade off implementation details. Three classes of quality are defined which differ in the filter-band attenuation (band overlap). Class 0 being the best, class 2 the worst acceptable. Furthermore two variants are offered regarding filter-frequency

bands, base ten: $10^{\frac{x}{10}}$ and base two: $2^{\frac{x}{3}}$. The centre frequency in either case is 1KHz, with (at least) 13 bands above and 16 bands below.

In the digital domain various alternative implementations are possible, most notably FFT and signal convolution approaches[3].

FFT (Fast Fourier Transform, an implementation of the discrete Fourier transform) transforms an audio signal from the time into the frequency domain. In the basic common form frequency bands are equally spaced and operation mode produces a flat response for white noise.

For musical applications a variant called 'perceptual analysers' is widespread. The signal level or power is weighted depending on various factors. Perceptual analysers often feature averaging functions or make use of screen-persistence to improve readability. They also come with additional features such as numeric readout for average noise level and peak detection to mitigate effects introduced by variation in the actual display.

2.9 Oscilloscopes

The oscilloscope is the "jack of all trades" of electronic instrumentation tools. It produces a two-dimensional plot of one or more signals as a function of time.

It differs from a casual wave-form display, which is often found in audio-applications, in various subtle but important details: An oscilloscope allows reliable signal measurement and numeric readout. Digital wave-form displays on the other hand are operating on audio-samples - as opposed to a continuous audio-signal. Figure 7 illustrates this.

For an oscilloscope to be useful for engineering work it must be calibrated - for both time and level, be able to produce an accurate readout of at least two channels and facilitate signal acquisition of particular events (triggering, signal history) [Adriaensen, 2013].

3 Standardisation

The key point of measuring things is to be able to meaningfully compare readings from one meter to another or to a mathematically calculated value. A useful analogy here is inches and centimetres, there is a rigorous specification of

[3]There are analogue designs to perform DFT techniques, but for all practical purposes they are inadequate and not comparable to digital signal processing.

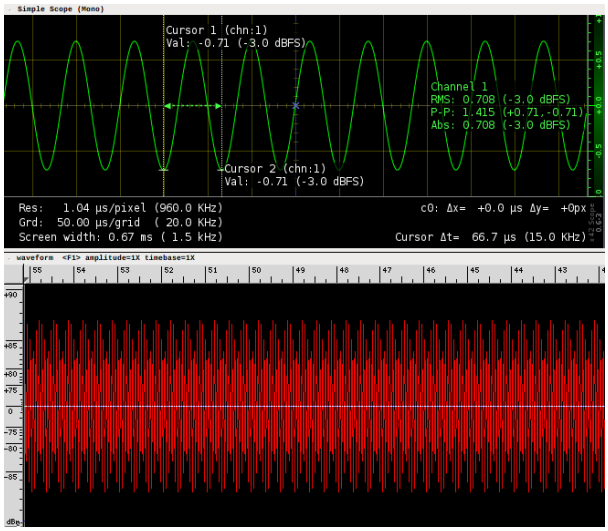


Figure 7: 15KHz, -3dBFS sine wave sampled at 48KSPS. The Oscilloscope (top) up-samples the data to reproduce the signal. The wave-form display (bottom) displays raw sample data.

what distance means. There are various standards and conventions, but there is no margin for error: One can rely on the centimetre.

Unfortunately the same rigour is not always applied to audio metering. On many products the included level meter mainly serves to enhance aesthetics, "make it look cool", rather than provide a reliable measurement. This trend increased with the proliferation of digital audio plugins. Those meters are not completely without merit, they can be useful to distinguish the presence, or otherwise, of a signal, and most will place the signal-level in the right *ballpark*. There is nothing wrong with saying "the building is tall" but to say "the building is 324.1m high" is more meaningful. The problem in the audio-world is that many vendors add false numeric labels to the scale to convey the look of professionalism, which can be quite misleading.

In the audio sphere the most prominent standards are the IEC and ITU specifications: These specs are designed such that all meters which are compliant, even when using completely different implementations, will produce identical results.

The fundamental attributes that are specified for all meter types are:

- Alignment or Reference Level and Range
- Ballistics (rise/fall times, peak-hold, burst response)
- Frequency Response (filtering)

Standards (such as IEC, ITU, EBU,...) govern many details beyond that, from visual colour indication to operating temperatures, analogue characteristics, electrical safety guidelines, test-methods, down to electrostatic and magnetic interference robustness requirements.

4 Software Implementation

4.1 Meters.lv2

Meters.lv2 [Gareus, 2013a] is a set of audio plugins, licensed in terms of the GPLv2 [GPL, 1991], to provide professional audio-signal measurements according to various standards. It currently features needle style meters (mono and stereo variants) of the following

- IEC 60268-10 Type I / DIN
- IEC 60268-10 Type I / Nordic
- IEC 60268-10 Type IIa / BBC
- IEC 60268-10 Type IIb / EBU
- IEC 60268-17 / VU

An overview is given in Fig. 2. Furthermore it includes meter-types with various appropriate visualisations for:

- 30 Band 1/3 octave spectrum analyser according to IEC 61260 (see Fig. 6)
- Digital True-Peak Meter (4x Oversampling), Type II rise-time, 13.3dB/s falloff.
- EBU R128 Meter with Histogram and History (Fig. 3)
- K/RMS meter, K-20, K-14 and K-12 variants
- Stereo Phase Correlation Meter (Needle Display, bottom right in Fig. 2)
- Goniometer (Stereo Phase Scope) (Fig. 8)
- Phase/Frequency Wheel (Fig. 4)

There is no official standard for the Goniometer and Phase-Wheel, the display has been eye-matched by experienced sound engineers to follow similar corresponding hardware equivalents.

Particular care has been taken to make the given software implementation safe for professional use. Specifically real-time safety and robustness (e.g. protection against denormals or subnormal input). The graphical display makes use of hardware acceleration (openGL) to minimise CPU usage.

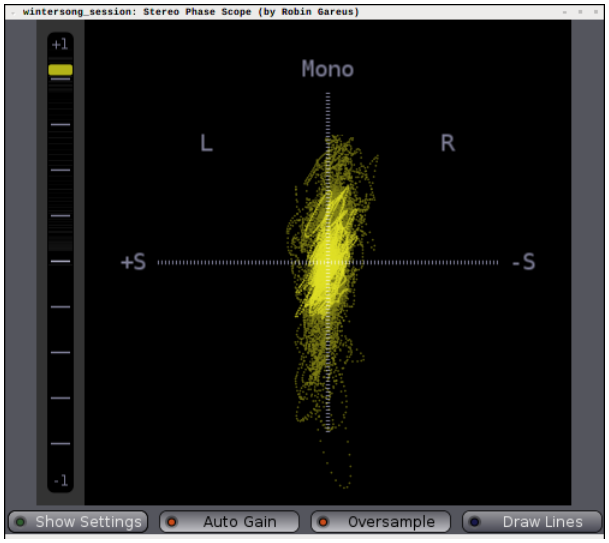


Figure 8: Goniometer (Phase Scope)

4.2 Sisco.lv2

Sisco.LV2 [Gareus, 2013c] implements a classic audio oscilloscope with variable time scale, triggering, cursors and numeric readout in LV2 plugin format. While it is feature complete for an audio-scope, it is rather *simplistic* compared to contemporary hardware oscilloscopes or similar endeavours by other authors [Adriaensen, 2013].

The minimum grid resolution is 50 microseconds - or a 32 times oversampled signal. The maximum buffer-time is 15 seconds. Currently variants up to four channels are available.

The time-scale setting is the only parameter that directly affects data acquisition. All other parameters act on the display of the data only. The vertical axis displays floating-point audio-sample values with the unit [-1..+1]. The amplitude can be scaled by a factor of [-10..+10] (20dB), negative values will invert the polarity of the signal. The numeric readout is not affected by amplitude scaling. Channels can be offset horizontally and vertically. The offset applies to the display only and does not span multiple buffers (the data does not extend beyond the original display). This allows the display to be adjusted in 'paused' mode after sampling a signal.

The oscilloscope allows for visually hiding channels as well as freezing the current display buffer of each channel individually. Regardless of display, data-acquisition for every channel continues and the channel can be used for triggering.

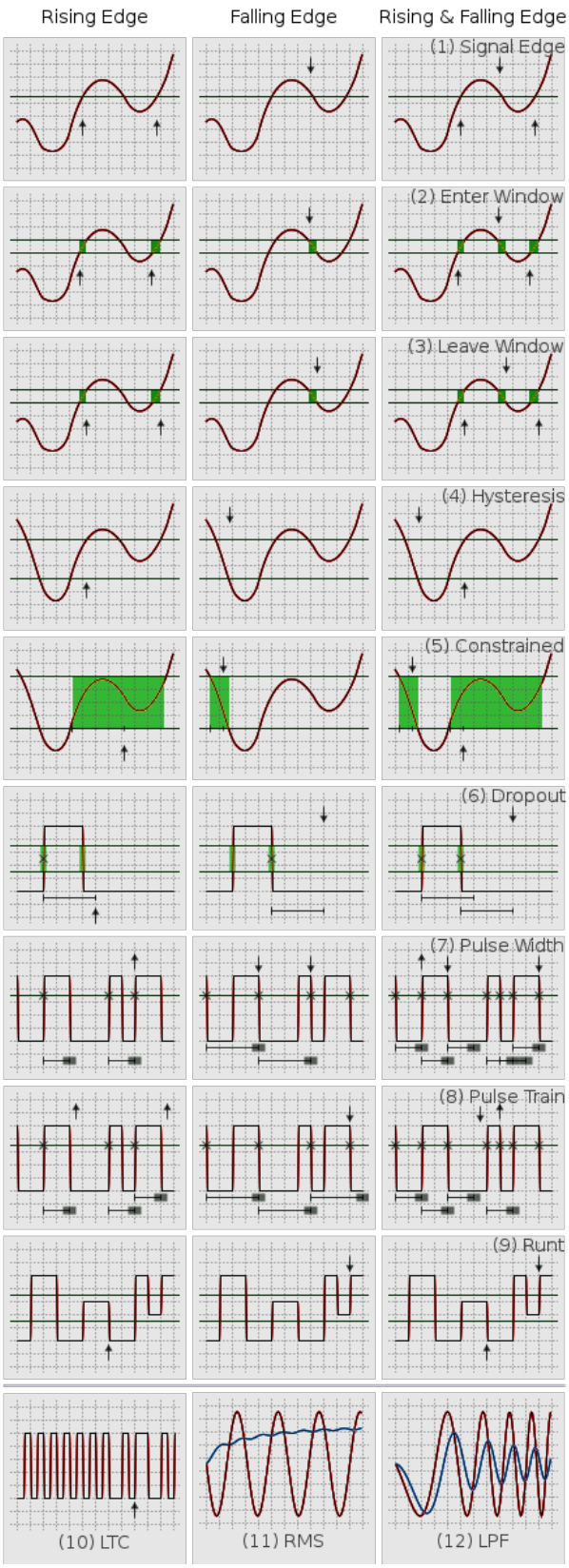


Figure 9: Overview of trigger preprocessor modes available in "mixtri.lv2". The arrow indicates trigger position.

#	Title	Description
1	Signal Edge	Signal passes 'Level 1'
2	Enter Window	Signal enters a given range (Level 1, 2).
3	Leave Window	Signal leaves a given range (Level 1, 2).
4	Hysteresis	Signal crosses both min and max (Level 1,2) in the same direction without interruption.
5	Constrained	Signal remains within a give range for at least 'Time 1'.
6	Drop-out	Signal does not pass through a given range for at least 'Time 1'.
7	Pulse Width	Last edge-trigger occurred between min and max (Time 1,2) ago.
8	Pulse Train	No edge-trigger for a give time (max, Time 2), or more than one trigger since a give time (min, Time 1).
9	Runt	Fire if signal crosses 1st but not 2nd threshold.
10	LTC	Trigger on Linear Time Code sync word.
11	RMS	Calculate RMS, Integrate over 'Time 1' samples.
12	LPF	Low Pass Filter, 1.0/ 'Time 1' Hz

Table 2: Description of trigger modes in Fig. 9.

The scope has three modes of operation:

- **No Triggering** The Scope runs free, with the display update-frequency depending on audio-buffer-size and selected time-scale. For update-frequencies less than 10Hz a vertical bar of the current acquisition position is displayed. This bar separates recent data (to the left) and previously acquired data (to the right).
- **Single Sweep** Manually trigger acquisition using the push-button, honouring trigger settings. Acquires exactly one complete display buffer.
- **Continuous Triggering** Continuously triggered data acquisition with a fixed hold time between runs.

Advanced trigger modes are not directly included with the scope, but implemented as a standalone "trigger preprocessor" [Gareus, 2013b] plugin, see Fig. 9 and Table 2. Trigger-modes 1-5 concern analogue operation modes, modes 6-9 are concerned with measuring digital signals[4]. Modes 10-12 are pre-processor modes rather than trigger modes. Apart from trigger and edge-mode selectors "mixtri.lv2" provides two level and two time control inputs for configuration.

5 Conclusion

An overview of various instrumentation tools and measurement standards was presented. The various tools are available as free software and have already found their way into GNU/Linux distributions, making Linux even more suitable as a platform for Pro-Audio work.

6 Acknowledgements

Kudos to Fons Adriaensen, DSP expert, author of jmeters, jkmeter, ebur128 and related zita-* software that was used as the basis to facilitate the software implementation of meters.lv2.

Thanks go to Jaromír Mikeš who created the initial software packages for Debian/Ubuntu, Alexandre Prokoudine who took great care of public relations, and Axel Müller who spent countless hours on testing and quality assurance.

References

Fons Adriaensen. 2011. Loudness measurement according to EBU R-128. In *Proceedings of the Linux Audio Conference 2011*.

Fons Adriaensen. 2013. Design of an audio oscilloscope application. In *Proceedings of the Linux Audio Conference 2013*, ISBN 9783-902949-00-4.

Eddy Brixen. 2010. *Audio Metering: Measurements, Standards and Practice*. ISBN 0240814673.

EBU. 2010. EBU R-128 – audio loudness normalisation & permitted maximum level. https://tech.ebu.ch/docs/r/r128.pdf. European Broadcast Union – Technical Committee.

EBU. 2011. EBU TECH 3342 – Loudness Range: A measure to supplement loudness normalisation in accordance with EBU R-128. https://tech.ebu.ch/docs/tech/tech3342.pdf. European Broadcast Union – Technical Committee.

Robin Gareus. 2013a. Meters.lv2 - audio plugin collection. https://github.com/x42/meters.lv2.

Robin Gareus. 2013b. Mixtri(x).lv2 - matrix mixer and trigger processor. https://github.com/x42/mixtri.lv2.

Robin Gareus. 2013c. Sisco.lv2 - Simple Scope. https://github.com/x42/sisco.lv2.

1991. Gnu general public license, version 2. https://www.gnu.org/licenses/old-licenses/gpl-2.0.html, June.

IEC. 1991. IEC60268-10 sound system equipment – Part 10 Peak programme level meters. International Electrotechnical Commission.

IEC. 1995. IEC61260 electroacoustics – Octave-band and fractional-octave-band filters. International Electrotechnical Commission.

ITU. 2006. ITU BS.1770, Algorithms to measure audio programme loudness and true-peak audio level. http://www.itu.int/rec/R-REC-BS.1770/en. International Telecommunication Union.

Bob Katz. 2000. How to make better recordings in the 21st century - an integrated approach to metering, monitoring, and leveling practices. http://www.digido.com/how-to-make-better-recordings-part-2.html. Updated from the article published in the September 2000 issue of the AES Journal by Bob Katz.

John Watkinson. 2000. *Art of Digital Audio*. ISBN 0240515870.

Wikipedia. 2013. Peak programme meter — wikipedia, the free encyclopedia. [Online; accessed 1-February-2014].

Wikipedia. 2014. VU meter — wikipedia, the free encyclopedia. [Online; accessed 2-February-2014].

[4] Digital signal trigger modes are of limited use with a generic audio interface, but can be useful in combination with an adapter (e.g. Midi to Audio) or inside Jack (e.g. AMS or ingen control signals).

Field Report on the OpenAV Release System

Harry VAN HAAREN
OpenAV Productions,
Mountshannon,
Co. Clare,
Ireland,
harryhaaren@gmail.com

Abstract

This paper discusses the OpenAV[1] release system, a new release system with at its core a balance between release date and financial support.

The release system works by creating the software, announcing it, and releasing after a waiting time. If money is donated to the project, the waiting time is reduced, which in turn results in an accelerated release.

This paper details the process of the OpenAV release system, discusses it in relation to other release systems. Finally the author draws on the experience gained by OpenAV Productions.

Keywords

Open-source, Funding, Software Release, OpenAV Productions

1 Introduction

This paper introduces the OpenAV release system, a release system which is designed to financially support the developer of a software project, while also ensuring that every project is released in source-code form.

Developers of open-source software often cannot work full time on a project due to financial constraints: they must earn money elsewhere in order to pay the bills. The OpenAV release system is designed to financially support a developer while working on an open source project.

The main components of the release system include a waiting time, a target amount for funding, and waiting before releasing source code. These components represent a balance, where both financial support and time passing contribute towards releasing the source code.

The outcome is always the same: the source code is released, the variable is how much money the developer received for their effort.

2 Background

In order to compare the OpenAV release system to existing funding and release-systems, a selection of well known crowd-funding projects are introduced below. Each section has a short introduction of the platform itself, and a description of its unique features.

2.1 Kickstarter

Kickstarter is a funding platform where projects are advertised, and can be donated to by members of the public. Project proposals are posted, usually with a video and blog post to gain momentum for the idea. The funding model is one of proposing an idea, and then attempting to collect the full amount of money: "Funding on Kickstarter is all-or-nothing - projects must reach their funding goals to receive any money"[2].

2.2 OpenInitiative.com

OpenInitiative use a pay-per-item model where developers suggest work on a project or feature, and then users can contribute to each feature or project in order to have the work done.

The unique feature is that "the developer determines the delivery date when the project is finished. Users then have 14 days to validate the result or request corrections. The developer is paid only after validation by the users"[3].

2.3 Snowdrift.coop

Snowdrift.coop is a new method of funding projects, where contributors pay more money depending on how many others contribute to the same goal: "I'll donate more if more people join me" [4].

This leads to a funding model that grows along with the projects it supports, making it sustainable for long-term funding.

2.4 Subscriptions and Donations

Allowing donations and/or subscriptions may provide financial support for a developer. Quantities of donations will vary depending on the amount of users directly benefiting from a project.

The motivation for donating or subscribing to a project are often driven by morals: generally there is no promise of a direct change in releasing based on donations.

3 OpenAV Release System

The OpenAV release system is a release system geared to provide software to the linux-audio community, while also financially rewarding the developer for time spent developing software.

The unique feature of this release system is that a release is not achieved by donating a fixed amount of money: instead a tradeoff between time and money dictates when the release occurs.

3.1 Design decisions

When OpenAV Productions was set up, the author researched how it could be financially supported while also releasing open source code.

It became clear that a new funding and release system could be more appropriate for developing and financially supporting projects than the existing solutions (e.g. Kickstarter).

The concept of setting a trade-off between release time and financial support became the core of the OpenAV release system. The release date is financially supported, instead of the product.

The developer commits to making an open source release: regardless of financial support, which ensures that the work done will become available to the commons. At the same time, an initiative exists to financially support the developer for a project, as the release of the code will be accelerated when money is donated.

3.2 Procedure

The stages of the release process are presented, after which each stage is detailed.

- Creation: the project is developed to a 1.0 degree of features and testing.
- Announcement: demonstrates the project, what it's purpose is, and how to use it.
- Releasing: the projects source code is made available.

3.2.1 Creation

In the first stage of the OpenAV release system the developer writes the software. During this stage they have the option to publicly consulte the community about the project if they so wish.

On completion of the features for a 1.0 release, testing is performed to verify the software is stable. When testing OpenAV software a group of trusted users are provided with the source code, and requested to not re-share the code. They can then use the software, and report bugs that were encountered.

When testing of the code has completed, the project is announced.

3.2.2 Announcement

In the announcement the developer demonstrates the software, what its purpose is, and what its features are. A good announcement makes it obvious to the audience of readers how they would benefit from the available of the project.

The announcement of the project includes two important factors for the release: the target amount and the waiting time. The target amount represents the amount of financial support the developer wishes to receive in return for creating the software. The waiting time is the amount of time that must pass before a release is made if no financial support is recieved.

The waiting time starts counting down from the date the announcement is made, and financial support in the form of donations is welcomed also from this date.

3.2.3 Releasing

The project is released when one of three situations occurs. These three situations are summarized, and then explained:

- The target amount of financial support is reached
- The waiting time expires, without financial support
- A combination of financial support and waiting time passing, as shown by: *Financial contributions + Waiting Time = Target Amount.*

Financial Support Target Reached

The target amount of money is reached by financial contributions. The developer has recieved the amount of financial support that they requested for an immediate release.

Waiting Time Expires

The waiting time for the project has passed: the project release is made without any financial contribution. The developer does not receive any financial support for their efforts.

Combination of Finance and Time
The target amount has been reached, partially by financial support, and partially by time passing. The developer has recieved some financial reward for their effort, and some time passed, adding up to the target amount.

Releasing
When any of the above three situations occur, the developer releases the source-code online allowing access to all.

4 Parties involved

This section details the point-of-view of the various parties involved in the OpenAV release system. Each party has specific positive and negative aspects with regards to their relation to the release system.

4.1 The Developer

When a developer uses the OpenAV release model, they create the environment for the production of software, both financially and for the code.

The software has to be written without financial support, as only after the projects announcement do they receive any financial support from it.

An announcement must be prepared, which shows off the features of the program. This is generally not necessary when releasing code, so could be considered extra work that the developer must do. Demonstrative videos or blog-style posts have been used by OpenAV Productions to publicize the software's functionality.

The previously prepared content must be broadcast to as large a user-base as possible: this involves using social-media extensively, writing emails to mailing lists, and posting on fora.

After completion of the project, the demonstrative content, and announcing it the developer waits for financial contributions. If contributions arrive, the release clock is updated, otherwise the waiting time is reduced according to the time passed.

4.2 The Contributor

When the OpenAV release system is in use, certain members of the community may decide to financially support the project. The donation accelerates the release of the project, but does not have any immediate result for the contributor.

As a return to the contributor the developer could list the contributors name, IRC nick or online handle on the project page to show their appreciation. OpenAV Productions lists contributors only after receiving a positive answer to the contributor being comfortable with such, and indicating their preferred name to be publicized. This is in order to maintain absolute privacy for contributors if they wish to remain anonymous.

4.3 The Library Developer

The authors of libraries that the project being released is based on make up this group of people. Although perhaps not directly involved in the OpenAV release model, the author feels it worth mentioning the library developers as an involved party as their code is in use by a project that is being financially supported by the community.

The project developer has the choice to donate some of the financial contribution they recieved to the library developer, however they are under no obligation to do so.

There is the possibility that a library developer doesn't agree with the release model which is being used by the project. Assuming that the license of code in question was not violated, one could say that it is irrelevant if the library developer doesn't agree with the release model: the license they chose is adhered to.

However, the fact that money is exchanged, and the library developers might not personally agree with the funding model is worth noting here.

4.4 The Remaining Community

The final "catch-all" group contains the community members who are not directly involved in the creation or funding of the project. Upon the release of the project, they gain source-access to the project too.

The fact that the whole community benefit from certain members financially supporting the developer is in the authors opinion the ultimate success of the OpenAV release system.

4.5 Statistics

This section introduces the statistics of the finances that OpenAV has recieved while working with the release system.

The data presented in table 1 shows details on the projects released by OpenAV Productions at time of writing. The columns show project title, hours spent developing the project, target

funding amount, waiting time, and number of days before the project was funded.

Project	Time	Target	Wait	Days
Sorcer	90	€120	1 year	9
Fabla	110	€120	1 year	8
ArtyFX	120	€120	1 year	5
Luppp	480	€520	1 year	5
ArtyFX 1.1	70	€120	1 year	8
Total	870	€1000	-	-

Table 1: Details of projects released by OpenAV Productions at time of writing.

4.6 Time

While developing the OpenAV projects the author has kept time spent developing. This was done using time tracker software, which provides breakdowns of time spent, and total hours.

Since the release of Sorcer in May, the total amount of time spent developing code for OpenAV productions is approx. 870 hours.

This figure does not include the development of Sorcer or Fabla (since they were started before Sorcer's release), however it does include work on some currently un-announced projects.

5 Discussion

This section discusses different aspects of the OpenAV release system. Various points-of-view are discussed with regards to the unique features of the OpenAV release system, compared to the other release systems presented in the background section.

5.1 Trust and Reputation

This section discusses the topic of trust between the developer and the community that applies to each funding model.

5.1.1 Code Quality

The OpenAV release model requires a basis of trust between the community and the developer. This trust in the developer takes its form as members of the community who financially contribute to the project believe that the quality and stability of the program is worth funding.

This trust can be built up over time by each developer by making smaller contributions, or releasing some code to prove their capabilities.

5.1.2 Target Funding

On announcement, the developer defines the target amount of financial support when using the OpenAV release system.

Contributors must make a decision when supporting a piece of software if they think the developers efforts are worth the finances they're asking for. This decision involves the contributors trust in the developers estimate of price, as well as their personal evaluation of the desirability of the resulting project.

5.1.3 Waiting Time and Funding

Upon announcing a project using the OpenAV release system, the developer must choose a waiting time before the project is released: even if no funding is recieved.

This waiting time is the tradeoff for financial contributions: a good balance between waiting time and target amount will motivate people to contribute to the project, because their contribution makes a significant improvement to the release date.

5.2 Release System Comparisons

This section discusses how the OpenAV release system compares with other release systems as presented in the background section: Kickstarter, OpenInitiative and Snowdrift.

5.2.1 Motivation for development

Kickstarter can be used to gain financial capital for commercial and closed source profit. It does not imply that the resulting software / project is released as open source.

On the contrary, the OpenAV release system incorporates a promise from the creator that the result of their work will be shared as open source regardless of the amount of funding that they may receive.

This fundamental difference between the funding motivation is one which is interesting to consider when discussing funding models for open source software.

5.2.2 Financial support

There are a variety of different choices to considering as to when a developer receives funding.

Kickstarter uses a "propose-fund-work" system which means that at worst the developer only makes a proposal, and if its not funded doesn't have to do any more work. OpenInitiative breaks this down into smaller stages, for a more finely-grained "propose-fund-work" system.

Snowdrift takes a totally different angle, and supports the developer financially without them having to make a commitment to a certain feature to develop.

The OpenAV release system takes a novel approach, which involves the developer doing all the work and then hoping to be financially supported for the time spent on that feature.

By using the OpenAV release system, a developer must be aware that they must do the initial development of the program without financial support.

5.2.3 Outcome of project

This section discusses the outcome of each project, based on the funding method used.

When funded with Kickstarter and OpenInitiative, if a proposal doesn't get funded then the work isn't completed. This means that the developer doesn't have to spend their own free time completing the work, but also that the community doesn't benefit from the work done.

Using the OpenAV release system, the developer takes on the risk of doing the work, and hoping to be financially supported later. In this way, the release model is more demanding for the developer, and less demanding of the community.

A positive aspect of the OpenAV release model is that the community can see the work done, and if they value it, they can contribute to the project in order to have it released sooner.

5.3 Financial Viability

This section deals with the financial viability of doing full time development of software using the OpenAV release system.

As presented in section *4.5 Statistics* of this paper , table 1 shows each project, the approximate amount of time spent on the project, and the amount of financial support recieved for the project.

Each project was released with 100% funding. This shows that the community are willing to financially support developers using this release model.

The hourly rate of pay is about €1.15. In order to make a living from releasing software by OpenAV, the target amounts would need to increase at least tenfold.

A tenfold increase in support would set the hourly rate at approx €12, which if worked for 40 hour weeks, 40 weeks a year, would result in a gross wage of €20,000.

The author feels that it is possible to achieve enough financial support to work full time on open source software using this release system.

6 Conclusion

This paper has presented the OpenAV release system, a new funding and release model that is geared towards small open source software projects.

A detailed procedure of how the OpenAV release system works is given. It was then discussed with regards to other funding and release systems, including Kickstarter, OpenInitiative and Snowdrift.

In the financial viability section the author draws from the experience gained from using the OpenAV release system for four software projects.

The author intends to continue using the OpenAV release model, perhaps one day being supported enough to work full time on open source projects.

7 Acknowledgments

Thanks to all financial contributors to OpenAV, all members of the linux audio community who have helped in designing features, testing code, and reporting bugs.

Thanks to all those who have contributed to the concept and philosophy behind the OpenAV release system, the conversations had have created the OpenAV release system.

Thank you all.

8 References

[1] http://www.openavproductions.com
[2] http://www.kickstarter.com/hello
[3] http://funding.openinitiative.com/presentation
[4] http://snowdrift.coop/p/snowdrift/w/slogan

Csound on the Web

Victor LAZZARINI and **Edward COSTELLO** and **Steven YI** and **John FITCH**
Department of Music
National University of Ireland
Maynooth,
Ireland,
{victor.lazzarini@nuim.ie, edwardcostello@gmail.com, stevenyi@gmail.com, jpff@codemist.co.uk }

Abstract

This paper reports on two approaches to provide a general-purpose audio programming support for web applications based on Csound. It reviews the current state of web audio development, and discusses some previous attempts at this. We then introduce a Javascript version of Csound that has been created using the Emscripten compiler, and discuss its features and limitations. In complement to this, we look at a Native Client implementation of Csound, which is a fully-functional version of Csound running in Chrome and Chromium browsers.

Keywords

Music Programming Languages; Web Applications;

1 Introduction

The web browser has become an increasingly viable platform for the creation and distribution of various types of media computing applications[Wyse and Subramanian, 2013]. It is no surprise that audio is an important part of these developments. For a good while now we have been interested in the possibilities of deployment of client-side Csound-based applications, in addition to the already existing server-side capabilities of the system. Such scenarios would be ideal for various uses of Csound. For instance, in Education, we could see the easy deployment of Computer Music training software for all levels, from secondary schools to third-level institutions. For the researcher, web applications can provide an easy means of creating prototypes and demonstrations. Composers and media artists can also benefit from the wide reach of the internet to create portable works of art. In summary, given the right conditions, Csound can provide a solid and robust general-purpose audio development environment for a variety of uses. In this paper, we report on the progress towards supporting these conditions.

2 Audio Technologies for the Web

The current state of audio systems for worldwide web applications is primarily based upon three technologies: Java[1], Adobe Flash[2], and HTML5 Web Audio[3]. Of the three, Java is the oldest. Applications using Java are deployed via the web either as Applets[4] or via Java Web Start[5]. Java as a platform for web applications has lost popularity since its introduction, primarily due to historically sluggish start-up times as well as concerns over security breaches. Also of concern is that major browser vendors have either completely disabled Applet loading or disabled them by default, and that NPAPI plugin support, with which the Java plugin for browsers is implemented, is planned to be dropped in future browser versions[6]. While Java sees strong support on the server-side and desktop, its future as a web-deployed application is tenuous at best and difficult to recommend for future audio system development.

Adobe Flash as a platform has seen large-scale support across platforms and across browsers. Numerous large-scale applications have been developed such as AudioTool[7], Patchwork[8], and Noteflight[9]. Flash developers can choose to deploy to the web using the Flash plugin, as well as use Adobe Air[10] to deploy to desktop and mobile devices. While these applications demonstrate what can be developed for the web

[1] http://java.oracle.com
[2] http://www.adobe.com/products/flashruntimes.html
[3] http://www.w3.org/TR/webaudio/
[4] http://docs.oracle.com/javase/tutorial/deployment/applet/index.html
[5] http://docs.oracle.com/javase/tutorial/deployment/webstart/index.html
[6] http://blog.chromium.org/2013/09/saying-goodbye-to-our-old-friend-npapi.html
[7] http://www.audiotool.com/
[8] http://www.patchwork-synth.com
[9] http://www.noteflight.com
[10] http://www.adobe.com/products/air.html

using Flash, the Flash platform itself has a number of drawbacks. The primary tools for Flash development are closed-source, commercial applications that are unavailable on Linux, though open source Flash compilers and IDEs do exist[11]. There has been a backlash against Flash in browsers, most famously by Steve Jobs and Apple[12], and the technology stack as a whole has seen limited development with the growing popularity of HTML5. At this time, Flash may be a viable platform for building audio applications, but the uncertain future makes it difficult to recommend.

Finally, HTML5 Web Audio is the most recent of technologies for web audio applications. Examples include the "Recreating the sounds of the BBC Radiophonic Workshop using the Web Audio API" site[13], Gibberish[14], and WebPd[15]. Unlike Java or Flash, which are implemented as browser plug-ins, the Web Audio API is a W3C proposed standard that is implemented by the browser itself.[16] Having built-in support for Audio removes the security issues and concerns over the future of plug-ins that affect Java and Flash. However, the Web Audio API has limitations that will be explored further below in the section on Emscripten.

3 Csound-based Web Application Design

Csound is a music synthesis system that has roots in the very earliest history of computer music. Csound use in Desktop and Mobile applications has been discussed previously in [Lazzarini et al., 2012b], [Yi and Lazzarini, 2012], and [Lazzarini et al., 2012a].

Prior to the technologies presented this paper, Csound-based web applications have employed Csound mostly on the server-side. For example, NetCsound [17] allows sending a CSD file to the server, where it would render the project to disk and email the user a link to the rendered file when complete. Another use of Csound on the server is Oeyvind Brandtsegg's VLBI Music [18], where Csound is running on the server and publishes its audio output to an audio stream that end users can listen to. A similar architecture is found in [Johannes and Toshihiro, 2013]. Since version 6.02, Csound also includes a built-in server, that can be activated through an option on start up. The server is able to receive code directly through UDP connections and compile them on the fly.

Using Csound server-side has both positives and negatives that should be evaluated for a project's requirements. It can be appropriate to use if the project's design calls for a single audio stream/Csound instance that is shared by all listeners. In this case, users might interact with the audio system over the web, at the expense of network latency. Using multiple realtime Csound instances, as would be the case if there was one per user, would certainly be taxing for a single server and would require careful resource limiting. For multiple non-realtime Csound instances, as in the case of NetCsound, multiple jobs may be scheduled and batch processed with less problems than with realtime systems, though resource management is still a concern.

An early project to employ client-side audio computation by Csound was described in [Casey and Smaragdis, 1996], where a sound and music description system was proposed for the rendering of network-supplied data streams. A possibly more flexible way to use Csound in client-side applications, however, is to use the web browser as a platform. Two attempts at this have been made in the past. The first was the now-defunct ActiveX Csound (also known as AXCsound)[19], which allowed embedding Csound into a webpage as an ActiveX Object. This technology is no longer maintained and was only available for use on Windows with Internet Explorer. A second attempt was made in the Mobile Csound Project[Lazzarini et al., 2012b], where a proof-of-concept Csound-based application was developed with Java and deployed using Java Web Start, achieving client-side Csound use via the browser. However, the technology required special permissions to run on the client side and required Java to be installed. Due to those issues and the unsure future of Java over the web,

[11] `http://www.flashdevelop.org/`

[12] `http://www.apple.com/hotnews/thoughts-on-flash/`

[13] `http://webaudio.prototyping.bbc.co.uk/`

[14] Available at `https://github.com/charlieroberts/Gibberish`, discussed in [Roberts et al., 2013]

[15] `https://github.com/sebpiq/WebPd`

[16] `http://caniuse.com/audio-api` lists current browsers that support the Web Audio API

[17] Available at `http://dream.cs.bath.ac.uk/netcsound/`, discussed in [ffitch et al., 2007]

[18] `http://www.researchcatalogue.net/view/55360/55361`

[19] We were unable to find a copy of this online, but one is available from the CD-ROM included with [Boulanger, 2000]

the solution was not further explored.

The two systems described in this paper are browser-based solutions that run on the client-side. The both share the following benefits:

- Csound has a large array of signal processing opcodes made immediately available to web-based projects.
- They are compiled using the same source code as is used for the desktop and mobile version of Csound. They only require recompiling to keep them in sync with the latest Csound features and bug fixes.
- Csound code that can be run with these browser solutions can be used on other platforms. Audio systems developed using Csound code is then cross-platform across the web, desktop, mobile, and embedded systems (i.e. Raspberry Pi, Beaglebone; discussed in [Batchelor and Wignall, 2013]). Developers can reuse their audio code from their web-based projects elsewhere, and vice versa.

4 Emscripten

Emscripten is a a project created by Alon Zakai at the Mozilla Foundation that compiles the assembly language used by the LLVM compiler into Javascript [Zakai, 2011]. When used in combination with LLVM's Clang frontend, Emscripten allows applications written in C/C++ or languages that use C/C++ runtimes to be run directly in web browsers. This eliminates the need for browser plugins and takes full advantage of web standards that are already in common use.

In order to generate Javascript from C/C++ sourcecode the codebase is first compiled into LLVM assembly language using LLVM's Clang frontend. Emscripten translates the resulting LLVM assembly language into Javascript, specifically an optimised subset of Javascript entitled asm.js. The asm.js subset of Javascript is intended as a low-level target language for compilers and allows a number of optimisations which are not possible with standard Javascript[20]. Code semantics which differ between Javascript and LLVM assembly can be emulated when accurate code is required. Emscripten has built-in methods to check for arithmetic overflow, signing issues and rounding errors. If emulation is not required, code can be translated without semantic emulation in order to achieve the best execution performance [Zakai, 2011].

Implementations of the C and C++ runtime libraries have been created for applications compiled with Emscripten. These allow programs written in C/C++ to transparently perform common tasks such as using the file system, allocating memory and printing to the console. Emscripten allows a virtual filesystem to be created using its FS library, which is used by Emscripten's libc and libcxx for file I/O[21]. Files can be added or removed from the virtual filesystem using Javascript helper functions. It is also possible to directly call C functions from Javascript using Emscripten[22]. These functions must first be named at compile time so they are not optimised out of the resulting compiled Javascript code. The required functions are then wrapped using Emscripten's *cwrap* function, and assigned to a Javascript function name. The *cwrap* function allows many Javascript variables to be used transparently as arguments to C functions, such as passing Javascript strings to functions which require the C languages *const char* array type.

Although Emscripten can successfully compile a large section of C/C++ code there are still a number of limitations to this approach due to limitations within the Javascript language and runtime. As Javascript doesn't support threading, Emscripten is unable to compile codebases that make use of threads. Some concurrency is possible using web workers, but they do not share state. It is also not possible to directly implement 64-bit integers in Javascript as all numbers are represented using 64-bit doubles. This results in a risk of rounding errors being introduced to the compiled Javascript when performing arithmetic operations with 64-bit integers [Zakai, 2011].

4.1 CsoundEmscripten

CsoundEmscripten is an implementation of the Csound language in Javascript using the Emscripten compiler. A working example of CsoundEmscripten can be found at `http://eddyc.github.io/CsoundEmscripten/`. The compiled Csound library and *CsoundObj* Javascript class can be found at `https://github.com/eddyc/CsoundEmscripten/`. CsoundEmscripten con-

[20] `http://asmjs.org/spec/latest/`

[21] `https://github.com/kripken/emscripten/wiki/Filesystem-API`

[22] `https://github.com/kripken/emscripten/wiki/Interacting-with-code`

sists of three main modules:

- The Csound library compiled to Javascript using Emscripten.
- A structure and associated functions written in C named *CsoundObj* implemented on top of the Csound library that is compiled to Javascript using Emscripten.
- A handwritten Javascript class also named *CsoundObj* that contains the public interface to CsoundEmscripten. The Javascript class both wraps the compiled *CsoundObj* structure and associated functions, and connects the Csound library to the Web Audio API.

4.1.1 Wrapping the Csound C API for use with Javascript

In order to simplify the interface between the Csound C API and the Javascript class containing the CsoundEmscripten public interface, a structure named *CsoundObj* and a number of functions which use this structure were created. The structure contains a reference to the current instance of Csound, a reference to Csound's input and output buffer, and Csound's 0dBFS value. Some of the functions that use this structure are:

- `CsoundObj_new()` - This function allocates and returns an instance of the *CsoundObj* structure. It also initialises an instance of Csound and disables Csound's default handling of sound I/O, allowing Csound's input and output buffers to be used directly.
- `CsoundObj_compileCSD(self, filePath, samplerate, controlrate, buffersize)` - This function is used to compile CSD files, it takes as its arguments: a pointer to the *CsoundObj* structure *self*, the address of a CSD file given by *filePath*, a specified sample rate given by *samplerate*, a specified control rate given by *controlrate* and a buffer size given by *buffersize*. The CSD file at the given address is compiled using these arguments.
- `CsoundObj_process(self, inNumberFrames, inputBuffer, outputBuffer)` - This function copies audio samples to Csound's input buffer and copies samples from Csound's output buffer. It takes as its arguments: a pointer to the *CsoundObj* structure *self*, an integer *inNumberFrames* specifying the number of samples to be copied, a pointer to a buffer containing the input samples named *inputBuffer* and a pointer to a destination buffer to copy the output samples named *outputBuffer*.

Each of the other functions that use the *CsoundObj* structure simply wrap existing functions present in the Csound C API. The relevant functions are:

- `csoundGetKsmps(csound)` - This function takes as its argument a pointer to an instance of Csound and returns the number of specified audio frames per control sample.
- `csoundGetNchnls(csound)` - This function takes as its argument a pointer to an instance of Csound and returns the number of specified audio output channels.
- `csoundGetNchnlsInput(csound)` - This function takes as its argument a pointer to an instance of Csound and returns the number of specified audio input channels.
- `csoundStop(csound)` - This function takes as its argument a pointer to an instance of Csound stops the current performance pass.
- `csoundReset(csound)` - This function takes as its argument a pointer to an instance of Csound and resets its internal memory and state in preparation for a new performance.
- `csoundSetControlChannel(csound, name, val)` - This function takes as its arguments: a pointer to an instance of Csound, a string given by *name*, and number given by *val*, it sets the numerical value of a Csound control channel specified by the string *name*.

The *CsoundObj* structure and associated functions are compiled to Javascript using Emscripten and added to the compiled Csound Javascript library. Although this is not necessary, keeping the compiled *CsoundObj* structure and functions in the same file as the Csound library makes it more convenient when including CsoundEmscripten within web pages.

4.1.2 The CsoundEmscripten Javascript interface

The last component of CsoundEmscripten is the *CsoundObj* Javascript class. This class provides the public interface for interacting with the compiled Csound library. As well as allocating an instance of Csound this class provides methods for controlling performance and setting the values of Csound's control channels. Additionally, this class interfaces with the Web Audio API, providing Csound with samples from the audio input bus and copying samples from Csound to the audio output bus. Audio I/O and the Csound process are performed in Javascript using the Web Audio API's *ScriptProcessorNode*. This node allows direct access to input and output samples in Javascript allowing audio processing and synthesis using the Csound library.

Csound can be used in any webpage by creating an instance of *CsoundObj* and calling the available public methods in Javascript. The methods available in the *CsoundObj* class are:

- `compileCSD(fileName)` This method takes as its argument the address of a CSD file *fileName* and compiles it for performance. The CSD file must be present in Emscripten's virtual filesystem. This method calls the compiled C function *CsoundObj_compileCSD*. It also creates a *ScriptProcessorNode* instance for Audio I/O.
- `enableAudioInput()` This method enables audio input to the web browser. When called, it triggers a permissions dialogue in the host web browser requesting permission to allow audio input. If permission is granted, audio input is available for the running Csound instance.
- `startAudioCallback()` This method connects the *ScriptProcessorNode* to the audio output and, if required, the audio input. The *ScriptProcessorNodes* audio processing callback is also started. During each callback, if required, audio samples from the *ScriptProcessorNodes* input are copied into Csound's input buffer and any new values for Csound's software channels are set. Csound's *csoundPerformKsmps()* function is called and any output samples are copied into the *ScriptProcessorNodes* output buffer.
- `stopAudioCallback()` This method disconnects the current running *ScriptProcessorNode* and stops the audio process callback. If required this method also disconnects any audio inputs.
- `addControlChannel(name, initialValue)` This method adds an object to a Javascript array that is used to update Csound's named channel values. Each object contains a string value given by *name*, a float value given by *initialValue* and additionally a boolean value indicating whether the float value has been updated.
- `setControlChannelValue(name, value)` This method sets a named control channel given by the string *name* to the specified number given by the *value* argument.
- `getControlChannelValue(name)` This method returns the current value of a named control channel given by the string *name*.

4.1.3 Limitations

Using CsoundEmscripten, it is possible to add Csound's audio processing and synthesis capabilities to any web browser that supports the Web Audio API. Unfortunately this approach of bringing Csound to the web comes with a number of drawbacks.

Although Javascript engines are constantly improving in speed and efficiency, running Csound entirely in Javascript is a processor intensive task on modern systems. This is especially troublesome when trying to run even moderately complex CSD files on mobile computing devices.

Another limitation is due to the design of the *ScriptProcessorNode* part of the Web Audio API. Unfortunately, the *ScriptProcessorNode* runs on the main thread. This can result in audio glitching when another process on the main thread—such as the UI—causes a delay in audio processing. As part of the W3Cs Web Audio Spec review it has been suggested that the *ScriptProcessorNode* be moved off of the main thread[23]. There has also been a resolution by the Web Audio API developers that they will make it possible to use the *ScriptProcessorNode* with web workers[24]. Hopefully in a future version of the Web Audio API the *ScriptProcessorNode* will be more capable of running the

[23] `https://github.com/w3ctag/spec-reviews/blob/master/2013/07/WebAudio.md#issue-scriptprocessornode-is-unfit-for-purpose-section-1`

[24] `https://www.w3.org/Bugs/Public/show_bug.cgi?id=17415#c94`

kind complex audio processing and synthesis capabilities allowed by the Csound library.

This version of Csound also does not support plugins, making some opcodes unavailable. Additionally, MIDI I/O is not currently supported. This is not due to the technical limitations of Emscripten, rather it was not implemented due to the current lack of support for the WebMIDI standard in Mozilla Firefox[25] and in the Webkit library[26].

5 Beyond Web Audio: Creating Audio Applications with PNaCl

As an alternative to the development of audio applications for web deployment in pure Javascript, it is possible to take advantage of the Native Clients (NaCl) platform[27]. This allows the use of C and C++ code to create components that are accessible to client-side Javascript, and run natively inside the browser. NaCl is described as a sandboxing technology, as it provides a safe environment for code to be executed, in an OS-independent manner [Yee et al., 2009] [Sehr et al., 2010]. This is not completely unlike the use of Java with the Java Webstart Technology (JAWS), which has been discussed elsewhere in relation to Csound [Lazzarini et al., 2012b].

There are two basic toolchains in NaCl: native/gcc and PNaCl [Donovan et al., 2010]. While the former produces architecture-dependent code (arm, x86, etc.), the latter is completely independent of any existing architecture. NaCl is currently only supported by the Chrome and Chromium browsers. Since version 31, Chrome enables PNaCl by default, allowing applications created with that technology to work completely out-of-the-box. While PNaCl modules can be served from anywhere in the open web, native-toolchain NaCl applications and extensions can only be installed from Google's Chrome Web Store.

5.1 The Pepper Plugin API

An integral part of NaCl is the Pepper Plugin API (PPAPI, or just Pepper). It offers various services, of which interfacing with Javascript and accessing the audio device is particularly relevant to our ends. All of the toolchains also include support for parts of the standard C library (eg. stdio), and very importantly for Csound, the pthread library. However, absent from the PNaCl toolchain are dlopen() and friends, which means no dynamic loading is available there.

Javascript client-side code is responsible for requesting the loading of a NaCl module. Once the module is loaded, execution is controlled through Javascript event listeners and message passing. A postMessage() method is used by Pepper to allow communication from Javascript to PNaCl module, triggering a message handler in the C/C++ side. In the opposite direction, a *message* event is issued when C/C++ code calls the equivalent PostMessage() function.

Audio output is well supported in Pepper with a mid-latency callback mechanism (ca. 10-11ms, 512 frames at 44.1 or 48 KHz sampling rate). Its performance appears to be very uniform across the various platforms. The Audio API design is very straightforward, although the library is a little rigid in terms of parameters. It supports only stereo at one of the two sampling rates mentioned above). Audio input is not yet available in the production release, but support can already be seen in the development repository.

The most complex part of NaCl is access to the local files. In short, there is no open access to the client disk, only to sandboxed filesystems. It is possible to mount a server filesystem (through httpfs), a memory filesystem (memfs), as well as local temporary or permanent filesystems (html5fs). For those to be useful, they can only be mounted and accessed through the NaCl module, which means that any copying of data from the user disk into these partitions has to be mediated by code written in the NaCl module. For instance, it is possible to take advantage of the file HTML5 tag and to get data from NaCl into a Javascript blob so that it can be saved into the user's disk. It is also possible to copy a file from disk into the sandbox using the URLReader service supplied by Pepper.

5.2 PNaCl

The PNaCl toolchain compiles code down to a portable bitcode executable (called a *pexe*). When this is delivered to the browser, an ahead-of-time compiler is used to translate the code into native form. A web application using PNaCl will contain three basic components: the pexe binary, a manifest file describing it, and a client-side script in JS, which loads and allows interaction with the module via the Pepper messaging

[25] `https://bugzilla.mozilla.org/show_bug.cgi?id=836897`

[26] `https://bugs.webkit.org/show_bug.cgi?id=107250`

[27] `https://developers.google.com/native-client`

system.

5.3 Csound for PNaCl

A fully functional implementation of Csound for Portable Native Clients is available from `http://vlazzarini.github.io`. The package is composed of three elements: the Javascript module (csound.js), the manifest file (csound.nmf), and the pexe binary (csound.pexe). The source for the PNaCl component is also available from that site (csound.cpp). It depends on the Csound and Libsndfile libraries compiled for PNaCl and the NaCL sdk. A Makefile for PNaCl exists in the Csound 6 sources.

5.3.1 The Javascript interface

Users of Csound for PNaCl will only interact with the services offered by the Javascript module. Typically an application written in HTML5 will require the following elements to use it:

- the csound.js script
- a reference to the module using a div tag with id="engine"
- a script containing the code to control Csound.

The script will contain calls to methods in csound.js, such as:

- `csound.Play()` - starts performance
- `csound.PlayCsd(s)` - starts performance from a CSD file *s*, which can be in ./http/ (ORIGIN server) or ./local/ (local sandbox).
- `csound.RenderCsd(s)` - renders a CSD file *s*, which can be in ./http/ (ORIGIN server) or ./local/ (local sandbox), with no RT audio output. The "finished render" message is issued on completion.
- `csound.Pause()` - pauses performance
- `csound.CompileOrc(s)` - compiles the Csound code in the string *s*
- `csound.ReadScore(s)` - reads the score in the string *s* (with preprocessing support)
- `csound.Event(s)` - sends in the line events contained in the string *s* (no preprocessing)
- `csound.SetChannel(name, value)` - sends the control channel *name* the value *value*, both arguments being strings.

As it starts, the PNaCl module will call a `moduleDidLoad()` function, if it exists. This can be defined in the application script. Also the following callbacks are also definable:

- `function handleMessage(message)`: called when there are messages from Csound (pnacl module). The string *message.data* contains the message.
- `function attachListeners()`: this is called when listeners for different events are to be attached.

In addition to Csound-specific controls, the module also includes a number of filesystem facilities, to allow the manipulation of resources in the server and in the sandbox:

- `csound.CopyToLocal(src, dest)` - copies the file *src* in the ORIGIN directory to the local file *dest*, which can be accessed at ./local/*dest*. The "Complete" message is issued on completion.
- `csound.CopyUrlToLocal(url,dest)` - copies the url *url* to the local file *dest*, which can be accessed at ./local/*dest*. Currently only ORIGIN and CORS urls are allowed remotely, but local files can also be passed if encoded as urls with the webkitURL.createObjectURL() javascript method. The "Complete" message is issued on completion.
- `csound.RequestFileFromLocal(src)` - requests the data from the local file *src*. The "Complete" message is issued on completion.
- `csound.GetFileData()` - returns the most recently requested file data as an ArrayObject.

A series of examples demonstrating this API is provided in github. In particular, an introductory example is found on `http://vlazzarini.github.io/minimal.html`.

5.3.2 Limitations

The following limitations apply to the current release of Csound for PNaCl:

- no realtime audio input (not supported yet in Pepper/NaCl)
- no MIDI in the NaCl module. However, it might be possible to implement MIDI in

JavaScript (through WebMIDI), and using the csound.js functions, send control data to Csound, and respond to the various channel messages.

- no plugins, as pNaCl does not support dlopen() and friends. This means some Csound opcodes are not available as they reside in plugin libraries. It might be possible to add some of these opcodes statically to the Csound pNaCl library in the future.

6 Conclusions

In this paper we reviewed the current state of support for the development of web-based audio and music applications. As part of this, we explored two approaches in deploying Csound as an engine for general-purpose media software. The first consisted of a Javascript version created with the help of the Emscripten compiler, and the second a native C/C++ port for the Native Client platform, using the Portable Native Client toolchain. The first has the advantage of enjoying widespread support by a variety of browsers, but is not yet fully deployable. On the other hand, the second approach, while at the moment only running on Chrome and Chromium browsers, is a robust and ready-for-production version of Csound.

7 Acknowledgements

This research was partly funded by the Program of Research in Third Level Institutions (PRTLI 5) of the Higher Education Authority (HEA) of Ireland, through the Digital Arts and Humanities programme.

References

Paul Batchelor and Trev Wignall. 2013. BeaglePi: An Introductory Guide to Csound on the BeagleBone and the Raspberry Pi, as well other Linux-powered tinyware. *Csound Journal*, (18).

Richard J. Boulanger, editor. 2000. *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press, February.

Michael Casey and Paris Smaragdis. 1996. Netsound. In *On the Edge*. ICMA and HKUST, August.

Alan Donovan, Robert Muth, Brad Chen, and David Sehr. 2010. PNaCl: Portable Native Client Executables. *Google White Paper*.

John ffitch, James Mitchell, and Julian Padget. 2007. Composition with sound web services and workflows. In Suvisoft Oy Ltd, editor, *Proceedings of the 2007 International Computer Music Conference*, volume I, pages 419–422. ICMA and Re:New, August. ISBN 0-9713192-5-1.

Tarmo Johannes and Kita Toshihiro. 2013. „Và, pensiero!“ - Fly, thought! Experiment for interactive internet based piece using Csound6 . `http://tarmo.uuu.ee/varia/failid/cs/pensiero-files/pensiero-presentation.pdf`. Accessed: February 2nd, 2014.

Victor Lazzarini, Steven Yi, and Joseph Timoney. 2012a. Digital audio effects on mobile platforms. In *Proceedings of DAFx 2012*.

Victor Lazzarini, Steven Yi, Joseph Timoney, Damian Keller, and Marco Pimenta. 2012b. The Mobile Csound Platform. In *Proceedings of ICMC 2012*.

Charles Roberts, Graham Wakefield, and Matthew Wright. 2013. The Web Browser As Synthesizer And Interface. *Proceedings of the International Conference on New Interfaces for Musical Expression*.

David Sehr, Robert Muth, Cliff Bife, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium*.

Lonce Wyse and Srikumar Subramanian. 2013. The Viability of the Web Browser as a Computer Music Platform. *Computer Music Journal*, 37(4):10–23.

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 IEEE Symposium on Security and Privacy*.

Steven Yi and Victor Lazzarini. 2012. Csound for Android. In *Linux Audio Conference*, volume 6.

Alon Zakai. 2011. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications*, pages 301–312. ACM.

BeaqleJS: HTML5 and JavaScript based Framework for the Subjective Evaluation of Audio Quality

Sebastian Kraft and **Udo Zölzer**
Department of Signal Processing and Communications
Helmut-Schmidt-University
Holstenhofweg 85, 22043 Hamburg, Germany
sebastian.kraft@hsu-hh.de

Abstract

Subjective listening tests are an essential tool for the evaluation and comparison of audio processing algorithms. In this paper we introduce BeaqleJS, a framework based on HTML5 and JavaScript to run listening tests in any modern web browser. This allows an easy distribution of the test environment to a significant amount of participants in combination with simple configuration and good expandability.

Keywords

listening test, subjective audio evaluation, HTML5, JavaScript

1 Introduction

Frequently used physical measures to judge the quality of audio signals, like the signal to noise ratio or signal distortion, do not correlate well with the perception of quality by the human hearing system. Therefore, listening tests, also named subjective audio evaluations, play an important role in the comparison of signal processing algorithms like audio effects and codecs.

The setup of a test environment and the selection of items under test is crucial to yield significant and non-biased results. Some guidance and standards can be found for example in the International Telecommunication Union (ITU) recommendations and in particular in [1]. Still, one of the biggest problems is to address an adequate number of qualified participants. Closely connected is the problem of distributing the test environment to the various platforms of the participants and how the results could be merged and evaluated afterwards.

In this paper BeaqleJS (browser based evaluation of audio quality and comparative listening environment) is presented, which is a framework to easily setup and run listening tests in any modern web browser. To achieve this, BeaqleJS purely relies on open web standards like HTML5 and JavaScript, without the need of further browser plugins or extensions. It is published under the GPLv3 open source license and its source code is available on GitHub[1].

The following section 2 will first introduce some background information about listening tests and common standards in general. Afterwards, the BeaqleJS framework is described in section 3 and section 4 outlines advanced usage scenarios like modifying or implementing new test schemes as well as server side evaluation and data collection. Finally section 5 will give a conclusion and outlook.

2 Listening test standards and basics

The difficulty in setting up a listening test comes from the fact that humans are rarely objective in their judgements. Therefore, the challenge is to design a test environment that minimizes external influences and yields non-biased results. To avoid mistakes it is advised to stick close to standardised instructions and test procedures as they are for example defined by the ITU in [1][2][3].

Test items should be presented in random order together with neutral names avoiding any association to the underlying algorithms. If several different algorithms are compared in one test, the corresponding items should always appear at a random position to prevent that the listeners recognize or learn the connection between a rating and its item position.

It is also necessary to find a way to judge the ability of the participants to understand the test procedure or to even recognize if they are able to perceive any differences between the items at all. For this purpose, a hidden reference and an anchor signal can be mixed among the test items. In valid test results, the participants should always rate the hidden reference with the same quality as the visible reference. In contrary, an anchor signal is an obviously bad test

[1] GitHub is a source code hosting platform using the git version control system http://www.github.com

item, for example heavily lowpass filtered, that is expected to always catch the worst rating and will set the bottom end of the scale.

The experience of the participants can have a strong influence on the results. People that are trained to hear analytically and know what artefacts they have to listen for can usually give more detailed feedback. On the other hand, they might be quite biased in their understanding of audio quality and typical consumers may highlight completely different aspects. Therefore, it makes sense to consider and document the background of the participants.

In a decentralized and distributed test setup it is important to assure comparable playback conditions. This is best achieved when all participants make use of high quality studio headphones as these completely reduce the influence of room acoustics and can be expected to be quite linear over a broad frequency range.

The selected audio test items should be well selected to reflect and underline a variety of characteristics of the tested algorithms. This can include for example transient, noisy and harmonic signals. A good starting point for choosing audio is the SQAM (Sound Quality Assessment Material) CD from the European Broadcasting Union (EBU) [4], which is well established in the audio coding field. The individual test item should be quite short and not exceed a length of 10 seconds. On the one hand, this helps to keep the attention of all listeners focused to the same part of the item, but also to avoid that exhaustion of the participants will influence the results. For the same reason, the amount of time that is necessary to perform the whole test should be kept below 15 minutes.

3 The BeaqleJS framework

BeaqleJS provides a framework to create browser based listening tests and is purely based on open web standards like HTML5 and JavaScript. For the user interface and to simplifiy Document Object Model (DOM) manipulations, the well known jQuery and jQueryUI libraries [5] are used.

The general structure of BeaqleJS can be divided in three blocks (Fig. 1). There is a common HTML5 `index.html` file to hold the main HTML structure with some basic place holder blocks whose content will be dynamically created by the JavaScript backend. The styling is completely independent and done with the help of cascading style sheets (CSS). Style sheets,

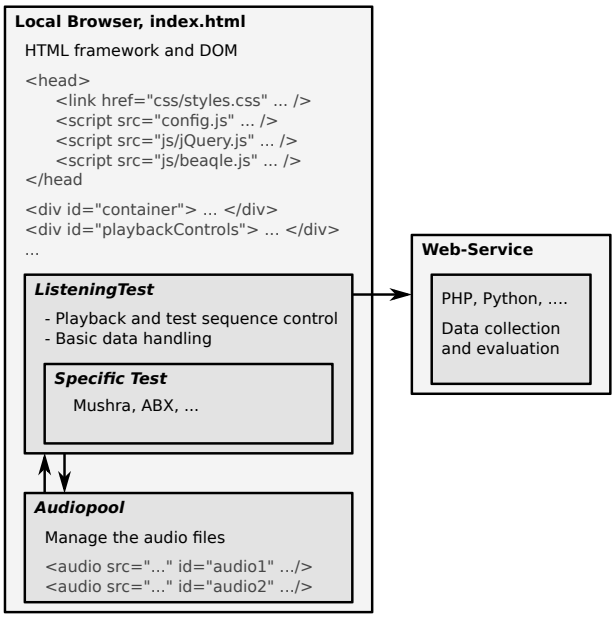


Figure 1: Schematic overview of the BeaqleJS framework.

config files and all necessary JavaScript libraries are loaded in the header of the `index.html`. Most of the descriptive text, like introduction and instructions, are placed in hidden blocks inside this file and their visibility is controlled by the scripts.

The JavaScript backend consists of two main classes. The first one is the `AudioPool` which takes care of audio playback and buffering. It pools a set of HTML5 <audio>-tags in a certain AudioPool -tag. There are simple functions to add and load a new file, connect and address it with an ID, manage playback and looping as well as synchronized pause and stop operations.

The `ListeningTest` class provides the main functions of an abstract listening test. This includes the setup and management of basic playback controls (play, pause, looping, time line display, ...), reading of the test configuration as well as storage of the results and also main control over the test sequence.

To create a certain test type the abstract `ListeningTest` class is inherited and specific functions for the actual arrangement of test items or storage and evaluation of the results need to be implemented (cf. section 4.1). Based on this modular approach it is very easy to extend the framework with additional test types or to create variants of existing ones without the need to reimplement all the necessary basics.

If the test is performed distributed over the

Table 1: Overview of supported codecs and audio formats in current web browsers.

Browser	Internet Explorer	Firefox	Chrome	Opera	Safari
WAV PCM	no	> 3.5	yes	> 11.00	> 3.1
Ogg Vorbis	no	> 3.5	yes	> 10.50	with XiphQT
MP3	> 9.0	> 26, not OS X	yes	> 14	> 3.1
AAC	> 9.0	> 26, not OS X	yes	> 14	> 3.1

internet or on several local computers, the `ListeningTest` main class is also able to send the final ratings to a web service for centralised collection and evaluation.

3.1 Codec support

Although the HTML5 markup language is already widely used in the internet there is no final adopted standard but only various drafts from the world wide web consortium (W3C). This may be one of the reasons why its degree of implementation can differ a lot between the different web browsers.

Our main interest regarding browser compatibility is the HTML5 <`audio`> element and its support for various file types, whereas in particular lossless formats like WAV or FLAC would be best suited for the desired application. An overview of the supported formats is given in Table 1 and unfortunately no browser supports FLAC[2] or other lossless codecs so far. The only lossless, but also uncompressed, format widely accepted is WAV PCM with 16 bit sample precision. Solely the Internet Explorer is not capable to play back this file type.

The described overall situation regarding the support of a common codec is quite unsatisfying. At the time of writing the only recommendation for an audio listening test environment would be to use the WAV PCM format. The circumstance that it is completely uncompressed (data rate approx. 94 kB/s per channel at 44.1 kHz sample rate) is relativised by the fact that the individual audio test items are recommended to be quite short, usually not more than 10 seconds, and therefore, the overall amount of audio data to be loaded is limited to around 1-2 MB per test item.

[2]It should be noted that there is a JavaScript based audio decoder framework named Aurora.js that is available together with a FLAC decoder at GitHub (https://github.com/audiocogs/aurora.js). However, its adaptability still has to be investigated.

3.2 Predefined tests

As described in the beginning of section 3 the main `ListeningTest` class only provides an abstract implementation with the core functionality of a generic listening test. Two implementations of specific listening tests are currently available in BeaqleJS. The most simple one is the so called ABX test and it is best suited to understand the functioning and internals of the whole framework. The other one is the so called MUSHRA (multi stimulus test with hidden reference and anchor) which is widely used in many evaluation scenarios and therefore, one of the most common test types. It is defined by the ITU in the BS.1534-1 recommendation [3].

3.2.1 ABX

In an ABX test (see Fig. 2) three items named A, B and X are presented to the listener, whereas X is randomly selected to be either the same as A or B. The listener has to identify which item is hidden behind X, or which one (A or B) is closest to X. If the listener is able to find the correct item, it reveals that there are perceptual differences between A and B.

A typical application of ABX tests would be the evaluation of the transparency of audio codecs. For example item A could be an unencoded audio snippet and B is the same snippet but encoded with a lossy codec. When the listener is not able to identify if A or B was hidden in X (results are randomly distributed), one can assume that the audio coding was transparent.

3.2.2 MUSHRA

In a MUSHRA test (see Fig. 3) the listener gets presented an item marked as reference together with several anonymous test items. By using a slider for each test item he has to rate how close the items are to the reference on top. Among the test items there is usually also one hidden reference and one, or several, anchor signals to prove the validity of the ratings and the quali-

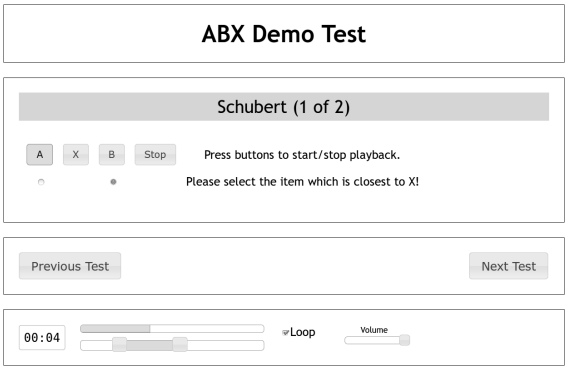


Figure 2: Screenshot of the ABX demo test.

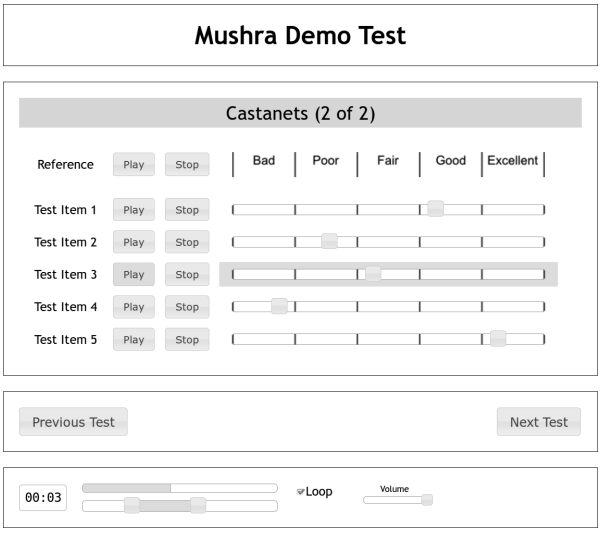


Figure 3: Screenshot of a MUSHRA test.

fication of the participants.

Contrary to ABX tests the MUSHRA procedure allows more detailed evaluations as it is possible to compare more than one algorithm to a reference. Furthermore, the results are on a continuous scale allowing a direct numerical comparison of all algorithms under test.

4 Advanced usage

If one of the predefined test classes already covers the desired test requirements, it is only necessary to create a set of test items and to define the corresponding paths in the config script. But it is also quite simple to slightly modify existing test layouts and structures or to even implement completely new test schemes.

4.1 Implementation of new tests

All new test classes have to inherit the base functionality from the main `ListeningTest` class. Inheritance in JavaScript is achieved by prototypes and this means to define a new class

```
// inherit from ListeningTest
function MyTest(TestData) {
  ListeningTest.apply(this, arguments);
}
MyTest.prototype = new ListeningTest();
MyTest.prototype.constructor = MyTest;

// implement the necessary functions
MyTest.prototype.createTestDOM = ...
MyTest.prototype.saveRatings   = ...
MyTest.prototype.readRatings   = ...
MyTest.prototype.formatResults = ...
```

Listing 1: Creation of a new test class `MyTest` inheriting from `ListeningTest`.

```
ListeningTest.TestState = {
// main public members
'CurrentTest': -1,
'TestIsRunning': false,
'FileMappings': {},
'Ratings': {},
'EvalResults': {},
// ...
// optionally add own fields
// ...
}
```

Listing 2: The `TestState` structure.

`MyTest` and then set its prototype to the base class. As this overwrites the constructor it has to be reset to the child constructor afterwards (listing 1). The child class can access the `TestState` (listing 2) and the `TestConfig` (listing 3) from the parent. The first one can be used to store random file mappings, ratings and other status variables, but can also be dynamically expanded with specific fields required by the child class. The `TestConfig` structure is just a mapping of the BeaqleJS config file into the class namespace. It has to contain at least the fields and structure as in listing 3 but it is possible to add additional sections which are then only read by the child class.

Every new test class has to implement at least four new functions:

- `createTestDOM(TestIdx)` creates the visible layout and HTML structure of the test with the index `TestIdx` based on the test configuration. All the necessary information from the config is available inside the object in `this.TestConfig.*`. Audio files should be appended to the `AudioPool` with `this.addAudio()` and can then be connected to play buttons by unique file IDs. Random mapping of file-

```
ListeningTest.TestConfig = {
"TestName": "Test",
"LoopByDefault": true,
"EnableABLoop": true,
"EnableOnlineSubmission": false,
"BeaqleServiceURL": "http://...",
"SupervisorContact": "super@visor.com",
"Testsets": [
    {
        "Name": "Testset 1",
        "Files": {
            // ...
        }
    },
    {
        "Name": "Testset 2",
        "Files": {
            // ...
        }
    }
    ],
// ...
// further test specific settings
// ...
}
```

Listing 3: The `TestConfig` structure.

names to IDs can be stored in the prepared `this.TestState.FileMappings[TestIdx]` structure (listing 2).

- `saveRatings(TestIdx)` is used to obtain the ratings from the sliders or buttons in the DOM and to store them in an arbitrary format in the predefined `this.TestState.Ratings[TestIdx].*` object.

- `readRatings(TestIdx)` is intended to read the ratings for `TestIdx` from `this.TestState.Ratings[TestIdx]` and to reapply them to the sliders or buttons in the DOM. This is primarily used during switching back and forth in the test sequence.

- `formatResults(TestIdx)` is automatically called after the final test in the sequence. It is supposed to evaluate and summarize the ratings and to store the final results in `this.TestState.EvalResults`. It should return a string containing the results formatted in a human readable manner (HTML). This will be presented to the listener after the last test and the `EvalResults` structure may be send to a web service (section 4.2).

4.2 Server side data collection and evaluation

Unfortunately it is not possible to directly send emails with JavaScript locally from a web browser or to store files. Therefore, to automatically collect the results it is necessary to have some kind of web service reachable from your network. This can be implemented for example with Python, Node.js or simply PHP.

The `ListeningTest` class includes the basic functionality to pack the results object `this.TestState.EvalResults` into a JSON (JavaScript Object Notation) structure and to transfer it to a web service for collection and further evaluation. A simple PHP example is included in the `beaqleJS_Service.php` file. It receives a JSON encoded data structure and writes its content into a text file with the time stamp as filename.

In the future, a more enhanced server side evaluation could include the automatic visualisation and statistical analysis of all the collected results. This can be combined with the capability to export the data in various formats for further analysis in scientific tools like SciPy, R or Matlab.

5 Conclusion

One big difficulty in setting up a proper listening test for the subjective evaluation of audio is its distribution to a significant number of participants. This is addressed by BeaqleJS which supplies all necessary components to run listening tests in any modern web browser in a flexible manner. It enables various usage scenarios ranging from complete online tests, over semi-public distribution in the intranet, down to local installation on a single computer with direct attendance of a supervisor. The presented framework, and its predecessor MushraJS, has already been used in various evaluations and proved its practical capabilities [6][7].

However, to assure significant and unbiased results, it is always advisable to closely stick to predefined and established test methods as they were introduced in section 2.

Further development could include a more extensive server side data evaluation and visualisation, but of course also the addition of more test schemes. The code is available at `https://github.com/HSU-ANT/beaqlejs` and the reader's contribution and feedback are highly appreciated.

References

[1] ITU-R. RECOMMENDATION ITU-R BS.1284-1: General methods for the subjective assessment of sound quality. Technical report, 2003.

[2] ITU-R. RECOMMENDATION ITU-R BS.1116-1: Methods for the subjective assessment of small impairments in audio systems including multichannel sound systems. Technical report, 1997.

[3] ITU-R. RECOMMENDATION ITU-R BS.1534-1: Method for the subjective assessment of intermediate quality level of coding systems (01/03). Technical report, 2003.

[4] European Broadcasting Union. Sound Quality Assessment Material recordings for subjective tests (SQAM), Audio CD. `https://tech.ebu.ch/publications/sqamcd`.

[5] The jQuery Foundation. Homepage of the jQuery and jQueryUI javascript libraries. `http://jquery.com/`.

[6] Sebastian Kraft, Martin Holters, Adrian von dem Knesebeck, and Udo Zölzer. Improved PVSOLA Time-Stretching and Pitch-Shifting for Polyphonic Audio. In *Proc. 15th Int. Conf. on Digital Audio Effects*, 2012.

[7] Marco Fink, Martin Holters, and Udo Zölzer. Comparison of Various Predictors for Audio Extrapolation. In *Proc. 16th Int. Conf. on Digital Audio Effects*, 2013.

Providing Music Notation Services over Internet

Mike SOLOMON, Dominique FOBER, Yann ORLAREY and **Stéphane LETZ**
Grame
Centre national de création musicale
Lyon - France
mike@mikesolomon.org, {fober, orlarey, letz}@grame.fr

Abstract

The GUIDO project gathers a textual format for music representation, a rendering engine operating on this format, and a library providing a high level support for all the services related to the GUIDO format and it's graphic rendering. The project includes now an HTTP server that allows users to access the musical-score-related functions in the API of the GUIDO-Engine library via uniform resource identifiers (URIs). This article resumes the core tenants of the REST architecture on which the GUIDO server is based, going on to explain how the server ports a C/C++ API to the web. It concludes with several examples as well as a discussion of how the REST architecture is well suited to a web-API that serves as a wrapper for another API.

1 Online musical editing

As client-server models for the processing, visualizing and analysis of data become more widespread in mobile computing (WordPress, YouTube, Instagram, SoundCloud), music engraving has entered the fray with various web-based score editing services. The GUIDO HTTP server merges the idea of a web-based music editor with a RESTful web service in order to expose the public API of the GUIDO-Engine library[Hoos and Hamel, 1997]. This section explores several categories of online musical editing services, concluding with a discussion of general trends in current technologies and the main problems that the tool outlined in this paper – the GUIDO HTTP server – seeks to address.

1.1 Online music notation editors

As of the writing of this paper (2014), there are three main online musical score editors – Noteflight[1], Melodus[2] and Scorio[3]. Noteflight and Melodus seek to provide a full-featured music editing platform online, similar to Google Documents' role in the world of office suites. Scorio is a hybrid tool that mixes rudimentary layout via a mobile editing platform with publication-quality layout via JIT compilation through LilyPond when possible.

1.2 Online score sharing software

Several music tools, such as Sibelius[4], MuseScore [Bonte, 2009], Maestro[5], and Capriccio[6], offer online services where scores composed using this software can be uploaded, browsed, and downloaded online. Capriccio, can be run online in limited form as a Java applet. MuseScore, Sibelius, and Maestro allow for automatic score/MIDI synchronisation of embedded files.

1.3 Online music JIT compilation services

WebLily[7], LilyBin[8], and OMET[9] are all JIT compilation services that run the LilyPond executable to compile uploaded code and return embedded SVG, canvas or PDF visualizations depending on the tool. The GUIDO note server [K. and Hoos, 1998] uses the GUIDOEngine library to compile Guido Music Notation Format [Hoos et al., 1998] strings into images.

1.4 A RESTful alternative

All of the tools described above facilitate the creation or visualization of scores via a variety of input methods (WYSIWYG, text, MusicXML

[1] http://www.noteflight.com
[2] http://www.melod.us
[3] https://scorio.com
[4] http://www.sibelius.com
[5] http://www.musicaleditor.com
[6] http://cdefgabc.com
[7] http://weblily.net
[8] http://www.lilybin.com
[9] http://www.omet.ca

etc.) but are not designed to facilitate low-latency server-client exchanges of score-related information. This is, in part, due to the fact that the majority of automated music engraving programs do not offer public APIs and are not designed to provide end-user information other than visual representations of scores and various non-human-readable file formats. The GUIDO Engine API [Daudin et al., 2009] [Grame, 2014b] seeks to remedy this issue by offering a public API that reports information about scores such as the number of pages, duration, and the placement of musical events both in time and on the page. The representational state transfer [Fielding, 2000], or REST, architectural style, is well suited for the porting of an API to the web because it is optimized for a system that is *stateless*, meaning that it does not require remembering intermediary states of a user. Contrast this to, for example, a server that needs to retain an undo history or the state of a logged-on user. As a result, the design of the server is clearer, quick and easy to scale [Richardson and Ruby, 2008]. This is further discussed in Section 3 and Section 4. The GUIDO HTTP server thus fills a gap in online score editing technology similar to the gap filled by Atom web feeds in news services.

2 Representational state transfer

Representational state transfer [Fielding, 2000] is an ubiquitous contemporary server architecture style [Richardson and Ruby, 2008]. The REST architecture is intended as a set of constraints to facilitate exchange in systems that deliver and report on hypermedia resources. The architectural style is based on a traditional *client-server* model with the design tradeoff that the server is *stateless*, meaning that all of the information required to process a request is contained in the request itself and the server does not need to store intermediary states. In order to speed up interaction with the server, the REST architecture calls for *client-side caching* of data, which can potentially eliminate certain redundant server requests. It also calls for a *uniform interface*, harmonizing all applications' interactions with the server at the expense of application-specific interaction models that could speed up exchanges. *Layering* is possible in this model, with intermediary servers translating various forms of shorthand into longer or less human-readable server commands. With this layering comes the constraint that exchanging agents cannot "see" beyond the layer with which they are communicating. As the burden on the client to be server-compliant is high in REST, the architectural style provides an optional constraint of servers' offering downloadable *code-on-demand* (scripts, applets, etc.) to ease client-side software development.

Certain specific architectural elements are put into place in order to facilitate the above-described architecture. In addition to the transferring of *data*, REST calls for the transferring of *meta-data* about a server response. This allows for the client side to have information about how to de-encode the response without needing to send specific de-encoding instructions. REST also encourages resource requests that are constructed in a hierarchical and human-readable manner. For example, accessing today's weather in Lyon, France is preferably

```
http://website.fr/France/Lyon/weather/today
```

rather than

```
http://website.fr/?country=France&town=Lyon
&feature=weather&date=today
```

A server compliant with the REST architecture is said to be a RESTful server.

3 The GUIDO HTTP server : an overview

The GUIDO Hypertext Transfer Protocol Daemon (HTTP) server is a RESTful server that compiles strings written in the GUIDO Music Notation (GMN) Format into musical scores and reports to the client several representations of this data.[10] It accepts user requests via two main methods of the HTTP protocol: POST, used to place elements on the server, and GET, used to retrieve information about elements on the server.

3.1 The POST method

POST, as implemented by the GUIDO server, is RESTful insofar as it does not save any information about the user state and only saves information sent by the user.

Assuming that a GUIDO HTTP server is running on the subdomain `http://guido.grame.fr` on port 8000, a POST request containing GMN code `[a b c d]` is sent via `curl` as follows:

```
curl -d"data=[a b c d]" http://guido.grame.fr:8000
```

[10] In this paper, the terms "GMN" and "score" are used interchangeably when talking about music treated by or stored on the server.

Assuming that the GMN code is valid, response, in JSON, gives the user a unique identifier generated using an SHA-1 tag corresponding to the input file. This ensures that the server will not store the same information multiple times:

```
{
 "ID": "07a21ccbfe7fe453462fee9a86bc806c8950423f"
}
```

This identifier is generated via the SHA-1 cryptographic hashing algorithm [Gallagher, 2012] that encodes any digital document as a 160-bit hash or key. The algorithm has a low incidence of collision ($\frac{1}{2^{63}}$), making it almost impossible for two documents to share the same SHA-1 key.

This is the server's internal representation of the GMN code and used for all subsequent requests to the server. To access it, it is appended onto the URI. The following is a simple request using the SHA-1 tag (hereafter shortened to facilitate readability) that results in the image seen in Figure 1.

```
curl http://guido.grame.fr:8000/07a21...0423f
```

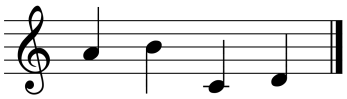


Figure 1: Score with SHA-1 tag `07a21...0423f`.

Technically speaking, the need to use an SHA-1 key in order to access scores and score-related information is not strictly RESTful. A strictly RESTful implementation would embed the score in every GET request. In accepting a GMN score via POST, the server must "remember" the score, which violates the principle of statelessness. The posting of a resource on the server is generally considered an acceptable compromise [Richardson and Ruby, 2008] so long as it is uniquely identifiable in an URI and the resource cannot be modified once uploaded on the server. This is the case with scores on the GUIDO server.

3.2 The GET method

Requests sent via GET query the server for information about scores. The main return type is JSON for all queries related to information about a score, MIDI for midi realizations of the score, and PNG for all queries asking for visual representations of the score itself. The latter is also possible in JPEG and SVG. All return types are specified in meta-data as per REST guidelines (see Section 2).

3.3 Uniform interface

The RESTful style specifies that a server's interface must be uniform, meaning that the operations that it executes must be the same for all clients interacting with the server. Furthermore, these operations should be conceptually different with no overlap and should ideally be widely used. The HTTP standard provides several atomic options that allow for the uniform interaction with a server [Richardson and Ruby, 2008]. The GUIDO web API uses the GET and POST methods from HTTP via libmicrohttpd [Grothoff, 2014], leaving out less widely-used methods such as PUT and DELETE in an effort to expose its full functionality to the largest group of client applications possible.

4 The GUIDO HTTP server as an API

The GUIDO HTTP server attempts to expose as much of the public API of the GUIDO Engine as possible, implementing one-to-one equivalencies with its functions when possible. Arguments are passed to these functions via optional key-value pairs in the URI's query part. Defaults are provided for all key-value pairs in case of omission. An exhaustive overview of the API can be found in the GUIDO HTTP server's documentation[Grame, 2014a].

This section aims to discuss some of the broad decisions made in exposing a C++ API via a web interface, giving three exhaustive examples at the end showing how the API is exposed.

4.1 SHA-1 key as musical score

Section 3.1 entertains the manner in which SHA-1 keys replace GMN scores in URIs sent to the server via in order to avoid having to send GMN scores in GET requests. This key corresponds to both an `ARHandler`, or *Abstract Representation*, and `GRHandler`, or *Graphic Representation* of a score in the GUIDO API. These two structures are used in order to generate information about the musical contents of a score (`ARHandler`) as well as its layout (`GRHandler`). The representation of both structures by one SHA-1 key allows the user to have a unique point of entry for each GMN score that conflates the data generated by several structures.

4.2 Function as URI segment

A function in the GUIDO public API is represented as a segment of the URI sent to the server. For example, the function

C/C++ API	URI segment	scope
GuidoGetPageCount	pagescount	score
GuidoGetVoiceCount	voicescount	score
GuidoDuration	duration	score
GuidoFindPageAt	pageat	score
GuidoGetPageDate	pagedate	score
GuidoGetPageMap	pagemap	score
GuidoGetSystemMap	systemmap	score
GuidoGetStaffMap	staffmap	score
GuidoGetVoiceMap	voicemap	score
GuidoGetTimeMap	timemap	score
GuidoAR2MIDIFile	midi	score
GuidoGetVersionStr	version	engine
GuidoGetLineSpace	linespace	engine

Table 1: GUIDO API public functions and their representations as URI segments.

`GuidoGetPageCount` in the GUIDO public API is represented as the URI segment `pagescount`.

The GUIDO public API provides two generic categories of functions:

- Functions addressed to the engine and reporting information about GUIDO.
- Functions addressed to a specific score processed by GUIDO.

With the C/C++ API, functions addressed to a score take *score handlers* as argument, which may be viewed as pointers to the internal score object. With HTTP, the SHA-1 tag plays the role of these score *score handlers* and the complete URI defines the scope of the request :

- Requests addressed to the engine are not prefixed.
- Requests addressed to a specific score are prefixed by the SHA-1 key.

For example,

```
http://guido.grame.fr:8000/version
```

reports the version of both GUIDO and the GUIDO server. On the other hand, the URI

```
http://guido.grame.fr:8000/<key>/voicescount
       where <key> is a SHA-1 key
```

exposes the API function `GuidoCountVoices` via the URI segment `voicescount`, giving the voice count of specific score.

Table 1 contains a succinct list of the servers' naming conventions showing the name of a function in the GUIDO public API, its representation as a server URI segment, and it's scope. Note that the only generic URI segment that does not correspond to a GUIDO public API function is `server`, which gives the version number of the server and thus is not related to the GUIDO API proper.

4.3 Arguments as key-value pairs

Several of the API functions listed in Table 1 require arguments in order to generate results. For example, the function `GuidoGetStaffMap` requires an argument `staff` specifying the staff for which the map should be generated. These arguments are specified in key-value pairs in the URI.

```
http://guido.grame.fr:8000/<key>/staffmap?staff=1
```

Default arguments are provided for all argument-taking functions in case the user fails to specify an argument. These arguments are values that would work in the majority of scores (for example, `page=1`) and often come from defaults provided in the API.

4.4 Layout and formatting options as key-value pairs

The GUIDO server allows for the specification of several parameters relating to the layout and formatting of scores as key-value pairs. These parameters are used in several different ways in the GUIDO public API. Some, such as `topmargin`, become values of structures such as `GuidoPageFormat`. Others, such as `resize`, represent calls to functions that effect layout (in this case `GuidoResizePageToMusic`). Yet others, such as `width`, are used at several points in the layout process depending on the chosen backend. Rather than devising separate URI construction conventions to represent different layout and formatting information in GUIDO, all layout and formatting options are implemented as key-value pairs to make interacting with the server uniform in keeping with RESTful style.

4.5 Return values

In order to handle the diversity of return types provided by the GUIDO API, the server attempts to find MIME types that best approximate the values returned by API functions. Sometimes, there is a direct correspondance. For example, the formats of images returned by the GUIDOEngine library when compiled with Qt (JPEG, PNG and SVG) are all MIME types.

In many cases, the GUIDO API returns custom structures that have no MIME type equivalent. In these cases, JSON [Crockford, 2013] is used to represent hierarchical relationships contained within these structures.

For example, the `Time2GraphicMap` struct is a composite structure consisting of pairs of `TimeSegment` and `FloatRect` structures. `TimeSegment` corresponds to beginning and end of a musical event whereas `FloatRect` corresponds to its placement on the page. To represent these structures in server responses, the GUIDO server uses JSON where key-value pairs correspond to a structure's element's name and its value. An example of this is given in Section 4.6.3, `time` corresponds to a `TimeSegment` and `graph` corresponds to a `FloatRect`.

4.6 Examples

4.6.1 voicescount

The command `voicescount` returns the number of voices in a score. It exposes the GUIDO Engine API method `GuidoCountVoices`. For example, the request:

```
http://guido.grame.fr:8000/<key>/voicescount
```

yields the following result:

```
{
  "<key>": {
    "voicescount": 1
  }
}
```

where "<key>" is the SHA-1 key given by the URI.

4.6.2 pageat

The command pageat returns the page given a specific date, expressed as a rational number. It exposes the GUIDO Engine API method `GuidoFindPageAt`. For example, the request:

```
http://guido.grame.fr:8000/<key>/pageat?date=1/4
```

yields the following result:

```
{
  "<key>": {
    "page": 1,
    "date": "1/4"
  }
}
```

4.6.3 staffmap

The command staffmap returns a map of the space each element of a given staff takes up in 2D space (represented by a box) and time space (represented as an interval of rational numbers). It exposes the GUIDO Engine API method `GuidoGetStaffMap`. For example, the request:

```
http://guido.grame.fr:8000/<key>/staffmap?staff=1
```

yields the following result, abbreviated below to minimize its space on the page:

```
{
  "<key>": {
    "staffmap": [
      {
        "graph": {
          "left": 916.18,
          "top": 497.803,
          "right": 1323.23,
          "bottom": 838.64
        },
        "time": {
          "start": "0/1",
          "end": "1/4"
        }
      },
      .
      .
      .
      {
        "graph": {
          "left": 2137.33,
          "top": 497.803,
          "right": 2595.51,
          "bottom": 838.64
        },
        "time": {
          "start": "3/4",
          "end": "1/1"
        }
      }
    ]
  }
}
```

5 Conclusion

The GUIDO HTTP server uses RESTful architectural principles such as statelessness, a uniform interface and a separation of client-server functionality in order to provide low-latency information retrieval. Information corresponds to uploaded GMN scores, encoded as various MIME types and transmitted via the HTTP protocol. The server exposes the robust GUIDO Engine public API via an interface based on standardized URI construction. It is intended for use by various applications needing to visualize musical scores and process score-related data. It is especially well-suited as an alternative to embarking libraries or external applications in score processing software. As cloud computing and mobile human-computer interaction becomes more common, this form of data transmission and processing is increasingly necessary. The GUIDO HTTP server intends to fill this by following RESTful architectural recommendations that have proven successful in other

server-based services.

The GUIDO project is an open source project hosted by sourceforge[11]. The GUIDO HTTP server is running at

`http://guidoservice.grame.fr/`.

6 Acknowledgements

This work has been realized in the framework of the INEDIT project that is supported by the French National Research Agency [ANR-12-CORD-0009].

References

T. Bonte. 2009. MuseScore: Open source music notation and composition software. Technical report, Free and Open source Software Developers' European Meeting. `http://www.slideshare.net/thomasbonte/musescore-at-fosdem-2009`.

D. Crockford. 2013. The json data interchange format. Technical report, ECMA International, October.

C. Daudin, D. Fober, S. Letz, and Y. Orlarey. 2009. The Guido Engine - a toolbox for music scores rendering. In *Proceedings of the Linux Audio Conference 2009*, pages 105–111.

R. Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. thesis, University of California, Irvine.

P. Gallagher. 2012. Secure hash standard (shs). Technical report, National Institute of Standards and Technology, March.

Grame, 2014a. *Guido Engine Web API Documentation v.0.50*.

Grame, 2014b. *GuidoLib v.1.52*.

C. Grothoff. 2014. GNU libmicrohttpd: a library for creating an embedded http server. `http://www.gnu.org/software/libmicrohttpd/index.html`.

H. H. Hoos and K. A. Hamel. 1997. The GUIDO Music Notation Format Specification - version 1.0, part 1: Basic GUIDO. Technical report TI 20/97, Technische Universitat Darmstadt.

H. Hoos, K. Hamel, K. Renz, and J. Kilian. 1998. The GUIDO Music Notation Format - a Novel Approach for Adequately Representing Score-level Music. In *Proceedings of the International Computer Music Conference*, pages 451–454. ICMA.

Renz K. and H. Hoos. 1998. A Web-based Approach to Music Notation Using GUIDO. In *Proceedings of the International Computer Music Conference*, pages 455–458. ICMA.

L. Richardson and S. Ruby. 2008. *RESTful Web Services*. O'Reilly Media.

[11] `http://guidolib.sf.net`

From Faust to Web Audio: Compiling Faust to JavaScript using Emscripten

Myles Borins
Center For Computer Research in Music and Acoustics
Stanford University
Stanford, California
United States,
mborins@ccrma.stanford.edu

Abstract

The Web Audio API is a platform for doing audio synthesis in the browser. Currently it has a number of natively compiled audio nodes capable of doing advanced synthesis. One of the available nodes the "ScriptProcessorNode" allows individuals to create their own custom unit generators in pure JavaScript. The Faust project, developed at Grame CNCM, consists of both a language and a compiler and allows individuals to deploy a signal processor to various languages and platforms. This paper examines a technology stack that allows for Faust to be compiled to highly optimized JavaScript unit generators that synthesize sound using the Web Audio API.

Keywords

WebAudio, Faust, Emscripten, JavaScript, asm.js

1 Introduction

The Web Audio API, released in 2011, is "a high-level JavaScript API for processing and synthesizing audio in web applications."[1] Currently there are a number of natively compiled audio nodes within the API capable of doing various forms of synthesis and digital signal processing. One of the available nodes, the "ScriptProcessorNode", allows individuals to create their own custom unit generators in pure JavaScript, extending the Web Audio API.

While the concept of making interactive sound synthesis environments in the browser is quite exciting, many factors stop individuals from investing time into the Web Audio platform. Ignoring the constraints of a single threaded environment there appear to be two primary limitations when working with web audio: There has not yet been enough Signal Processing related JavaScript code written yet, and some signal processing concepts prove difficult to implement efficiently in a loosely typed language with no memory management.

[1] https://dvcs.w3.org/hg/audio/raw-file/tip/WebAudio/specification.html

2 Some Context

2.1 WAAX and Flocking

There are a number of projects that are in development abstracting over top of the Web Audio API in order to extend its capabilities, create more complicated unit generators, and allow for a more intuitive syntax. Projects such as WAAX (Web Audio API eXtension)[2] by Hongchan Choi do so while using only the natively compiled nodes in order to ensure optimum efficiency.[H. Choi and J.Berger, 2013]

While these projects offer a wide variety of unit generators and synthesis modules, they cannot be used to implement all cutting edge techniques. For example the delay node interface does not offer a tap in or tap out function, making wave guide models impossible to implement.[3]

The Flocking audio synthesis toolkit[4] by Colin Clark offers a unique declarative model for doing signal processing within the browser. Unlike WAAX, Flocking has opted to internally manage all signal generation and using a single "ScriptProcessorNode" to hand off precomputed buffers of samples.

WAAX and Flocking offer two very different approaches to Web Audio. WAAX offers efficiency, whereas Flocking offers an extensible architecture and declarative syntax in which web developers can write their own first-class custom unit generators. That being said both projects suffer from the same problem, a lack of man hours. There are only so many individuals who have the time and domain specific knowledge necessary to contribute to their development.

2.2 Introduction to Faust

The Faust project offers a unique solution to this problem; rather than write code, generate

[2] https://github.com/hoch/waax

[3] https://dvcs.w3.org/hg/audio/raw-file/tip/WebAudio/specification.html#DelayNode-section

[4] http://flockingjs.org/

it. Faust, developed at Grame CNCM, "is a programming language that provides a purely functional approach to signal processing while offering a high level of performance." [Orlarey et al., 2009] The project is both a language and a compiler, offering the ability to write code once and deploy to many different signal processing environments.

Faust also has a community of scientists and developers who have contributed a large amount of code waiting to be compiled to other platforms. For example Julius Smith has done a substantial amount of research using Faust to implement wave guide synthesis models[Smith et al., 2010] and Romain Michon has ported the entire STK to Faust.[Michon and Smith, 2011]

Creating an efficient compile path from Faust to the Web Audio API would allow for all of the available Faust code to immediately be able to run in the browser. Further, using the architecture compilation model that Faust is famous for we would be able to wrap the compiled Web Audio code to be compatible with all current libraries and frameworks such as WAAX and Flocking.

2.3 Current Web Audio Implementation

Currently there is an implementation done by Stéphane Letz to compile Faust to Web Audio directly from the Faust Intermediate Representation[5]. While the implementation is elegant, any algorithms relying on integer arithmetic are currently broken due to JavaScript representing all Numbers as 32-bit floating point at a binary level.

2.4 Introduction to asm.js

One way to do integer arithmetic with cross-browser support is asm.js. The asm.js specification[6] outlines a 'strict subset' of JavaScript that offers a unique programming model. Through the use of typed arrays[7] it is possible to do integer and floating-point arithmetic. This is done with a virtual machine that gives developers access to a heap and functions to be used to manage memory and perform arithmetic operations.

While it would have been possible to use Stéphane Letz's work as a starting point and extend the current WebAudio architecture to utilize the asm.js susbset, it would require quite a bit of overhead. Not only would integer and floating point specific interpretation need to be implemented, but a functional virtual machine would need to have been developed in order to take advantage of asm.js.

Further, we would not see any of the optimization benefits that one would get from a modern compiler such as gcc or clang. In the spirit of this project, a search was done to find a way to automate away the need to worry about all of these complications.

2.5 Introduction to Emscripten

Emscripten is a project started by Alon Zakai from Mozilla that compiles LLVM(Low Level Virtual Machine) assembler to JavaScript, specifically asm.js.[Zakai, 2011] The platform is both a compiler and a virtual machine capable of running C and C++ code in the browser.

Emscripten gives you an interface to break out C functions so that they can be called using JavaScript. It also provides functions for managing memory in the virtual machine your C code is running. These functions allow you to allocate new memory to be operated on (in the case of sound buffers), and the ability to manipulate memory in the heap (in order to change parameters).

Currently Faust is able to compile to a C++ file using the minimal.cpp architecture file, the resulting file can painlessly be compiled to asm.js with Emscripten. The upstream Faust2 branch can compile Faust to LLVM byte-code which offers another potential compilation path.

3 Making Noise

A first approach to automating the compilation process from Faust to Web Audio involves manually implementing each step. The Faust code needs to be compiled to C++ and have the resulting dsp class wrapped in order to allow internal data and member functions to be accessed once compiled to JavaScript. The resulting C++ file then needs to be compiled by Emscripten to asm.js. The asm.js needs to once again be wrapped in order to provide an intuitive JavaScript interface that will operate on the dsp object running in the Emscripten virtual machine. Finally an interface between the Emscripten virtual machine and WebAudio needs to be made to hand off samples that need to be sonified.

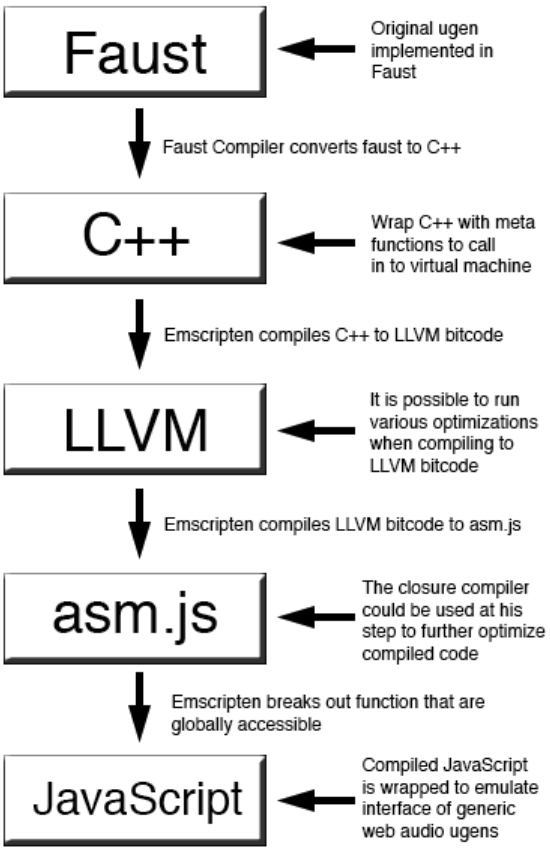
[5] http://faust.grame.fr/index.php/7-news/73-faust-web-art

[6] http://asmjs.org/spec/latest/

[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays?redirectlocale=en-US&redirectslug=JavaScript/Typed_arrays

As an initial proof of concept it was attempted to compile the example noise.dsp that comes shipped with Faust to JavaScript by way of Emscripten. Noise was a prime candidate for these initial tests due to the integer specific calculations used in its algorithm.

The below sections will describe the process used to manually implement noise in the browser starting from a faust dsp file, and ending with a working Web Audio API JavaScript Object.



3.1 Faust Source

The noise unit generator starts as a Faust dsp file.

```
random  = +(12345)~*(1103515245);
noise   = random/2147483647.0;
process = noise * 0.5;
```

In order to compile to C++ in a manner that will be compatible with Emscripten we must use the follow command.

```
faust -a minimal.cpp -i -uim \
-cn Noise  dsp/noise.dsp \
  -o cpp/faust-noise.cpp
```

This tells faust to compile the above code using the *minimal.cpp* architecture file, to call the object being created Noise, and to include all necessary header files and dependencies.

The resulting C++ code need to be wrapped with a series of meta functions that can be called to operate on objects living in the virtual machine. A constructor and destructor are implemented in order to create objects and properly clean them up, and a compute function is then used to grab the latest frame of samples from the unit generator. In order to change the state of the unit generator after its instantiated in the heap a number of other functions are available to create a map of the ugen's parameters, and get / set values.

3.2 Emscripten & asm.js

Once the wrapper has been concatenated with the Faust compiled C++ it can then be compiled by Emscripten to asm.js. This is done with the following command

```
emcc cpp/faust-noise.cpp -o \
  js/faust-noise-temp.js \
  -s EXPORTED_FUNCTIONS="\
  ['_NOISE_constructor',\
  '_NOISE_destructor',\
  '_NOISE_compute',\
  '_NOISE_getNumInputs',\
  '_NOISE_getNumOutputs',\
  '_NOISE_getNumParams',\
  '_NOISE_getNextParam']"
```

Note the exported functions, which are referencing the seven wrapper functions mentioned in the previous step. This is required to stop Emscripten from obfuscating the names of the functions when certain optimization flags are thrown during compilation, and to make access to them available in the global namespace of JavaScript.

3.3 Web Audio Api

Once the asm.js code has been compiled a JavaScript wrapper is used to break out the functionality of the code into JavaScript functions. As well, the correct context for generating audio in the browser needs to be set up within the Web Audio API, connecting the generated data from the Faust generated functions to the correct Web Audio API functions in order to generate sound. Again this wrapper can be found in the source repository on GitHub

4 Results

Using the above methods a Faust compiled WebAudio noise unit generator was successfully created. The result can be found at:

`http://thealphanerd.io/examples/faust2webaudio/`

4.1 Other Examples

This process has been repeated for a number of other unit generators including a sine oscillator, freeverb, and a 16th order FDN reverb (included in the Faust distribution as Reverb Designer). All three examples work in the browser, although the 16th order FDN takes a few seconds to get going. Once the unit generators have been compiled to JavaScript it is quite easy to connect them to each, and other web audio components.

Below is an example of how to create a noise object, and apply freeverb to its output. Both of these objects have been compiled using Faust2WebAudio. This example can be found online at `http://thealphanerd.io/examples/faust2webaudio/freeverb.html`

```
var noise = faust.noise();
var freeverb = faust.freeverb();
noise.connect(freeverb);
noise.update("Volume", 0.1);
freeverb.update("Damp", 0.75);
freeverb.update("RoomSiz", 0.75);
freeverb.update("Wet", 0.75);
freeverb.play();
```

5 Limitations

Currently the automation layer has not yet been completed. While the wrapper scripts have all been generically written, hand written bash scripts utilizing tools such as sed are currently being used to compile individual unit generators. A next step would involve moving the generic wrappers in to their own architecture file and relying on the Faust build system to handle generic compilation.

Another major limitation is that I am currently utilizing a separate instance of the Emscripten virtual machine for each unique unit generator. This is an unfortunate side effect of the current compilation method. Emscripten includes the virtual machine at the head of every compiled js file. There is an option to statically link a number of compiled js files to a single optimized file with redundancies removed, but I am concerned about the implications of that workflow.

A developer would be required to supply all of the faust objects at once, and not have the ability to swap in and out files at their leisure. Unless their is an intuitive and fast way to compile this final file, it will make it difficult for individuals to add new unit generators on the fly as they are composing in the browser.

One solution is to utilize a JavaScript task runner such as grunt to watch for changes in specific directories / files and to properly compile and statically link multiple files on the fly.

6 Looking Forward

While the above mentioned limitations do need to be worked on, benchmarks should be performed on the currently compiled code to ensure that this compilation method is in fact a good direction.

As well, Stéphane Letz and Yann Orley have expressed a desire to approach this problem using their original method of going directly from the Faust Intermediate Representation to JavaScript. This would avoid moving from a functional language to an object oriented language back to a function language, which has proven somewhat inelegant. It may prove appropriate once the Emscripten method can be benchmarked to put time in to developing this more direct compilation path so that the results from the two methods can be compared.

7 Conclusion

The results of this research have shown that it is indeed possible to get compiled Faust code running properly in the browser. This is very exciting, as if the benchmarks are encouraging we will be able to use the resulting code to greatly expand the ecosystem for digital signal processing in the browser.

One of the most exciting parts of the results are that if this process can be perfected we will continue to see improvements in efficiency as the various technologies we are relying on continue to improve. As JavaScript becomes more efficient, so does the compiled code. As WebAudio becomes more stable, so does the compiled code. As asm.js optimizations improve in the browser, we get the optimizations for free. Simply put, even if the resulting benchmarks prove to not be competitive with current hand written JavaScript, it will only get better with

time while requiring minimal time maintaing the project.

8 Code Repository

Find the source online at:
`https://github.com/TheAlphaNerd/faust2webaudio`

9 Acknowledgements

I would like to thank Julius O. Smith, Stéphane Letz, Yann Orley, and Colin Clark for their guidance and support.

References

H. Choi and J.Berger. 2013. Waax: Web audio api extension. In *Proceedings of the Thirteenth New Interfaces for Musical Expression Conference*.

R. Michon and J. O. Smith. 2011. Faust-stk: A set of linear and nonlinear physical models for the faust programming language. In *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11)*, pages 65–96, September.

Y. Orlarey, D. Fober, and S. Letz. 2009. Faust : an efficient functional approach to dsp programming. In *New Computational Paradigms for Computer Music*, pages 65–96. Editions DELATOUR FRANCE.

J. Smith, J. Kuroda abd J. Perng abd K. V. Heusen, and J. Abel. 2010. Efficient computational modeling of piano strings for real-time synthesis using mass-spring chains, coupled finite differences, and digital waveguide sections. In *Acoustical Society of America, Program of the 2nd Pan-American/Iberian Meeting on Acoustics (abstract and presentation)*, pages 65–96, Nov.

A. Zakai. 2011. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM.

The Ambisonic Decoder Toolbox: Extensions for Partial-Coverage Loudspeaker Arrays

Aaron J. HELLER
AI Center, SRI International
Menlo Park, CA, US
heller@ai.sri.com

Eric M. BENJAMIN
Surround Research
Pacifica, CA, US
ebenj@pacbell.net

Abstract

We present extensions to the Ambisonic Decoder Toolbox to efficiently design periphonic decoders for non-uniform speaker arrays such as hemispherical domes and multilevel rings. These techniques include modified inversion, AllRAD, and spherical Slepian function-based decoders. We also describe a new backend for the toolbox that writes out full-featured decoders in the FAUST DSP specification language, which can then be compiled into a variety of plug-in formats. Informal listening tests and performance measurements indicate that these decoders work well for speaker arrays that are difficult to handle with conventional design techniques. The computation is relatively quick and more reliable compared to non-linear optimization techniques used previously.

Keywords

Ambisonic decoder, HOA, hemisphere, FAUST

1 Introduction

This is a paper about extensions to the Ambisonic Decoder Toolbox to efficiently design decoders for loudspeaker arrays with partial coverage of the sphere, such as domes and multilevel rings. The criteria for Ambisonic reproduction are:

- Constant amplitude and energy gain for all source directions
- At low frequencies, reproduced wavefront direction and velocity are correct
- At high frequencies, maximum concentration of energy in the source direction
- Matching high- and low-frequency perceived directions

In the case of decoders for partial-coverage arrays, we relax these to apply only to source directions that are within the covered part of the sphere, but still require that the decoder be "well behaved" for sources from other directions.

Conventional techniques for periphonic decoder design work well when the speakers are distributed uniformly around the listening position. First-order Ambisonics can be accommodated in many listening rooms; however, when moving to higher-order reproduction the need arises to place more loudspeakers below the listener. This requires placing the listening position high in the room or on an acoustically transparent floor with a space below to install speakers. Neither of these are practical for most installations, so hemispherical dome configurations are a popular alternative. In addition, it may be impractical to install speakers directly overhead, resulting in a configuration of horizontal rings of speakers at multiple heights. These configurations leave gaps in coverage below, and possibly above, the listening position.

In a previous paper, we describe a MATLAB/GNU Octave[1] toolbox for generating Ambisonic decoders that uses inversion or projection to generate an initial estimate and then non-linear optimization to simultaneously maximize r_E and minimize directional and loudness errors [2012]. While this works well for small arrays, we found that increasing the Ambisonic order and number of loudspeakers causes the optimizer to converge slowly and get stuck in local minima unless the starting solution is close to optimal.[2]

In the case of hemispherical domes and multilevel rings, neither inversion or projection provide a close starting point. Once the speaker array deviates from uniform geometry, an inversion decoder will trade uniform loudness for directional accuracy by putting more energy in directions where gaps between the loudspeakers are larger. A projection decoder does just the opposite, putting equal energy into all the speak-

[1] In this paper, we use "MATLAB" to refer to both MATLAB and GNU Octave. Care has been taken to make sure the code runs in both; however, not all of the graphics work well in Octave. MATLAB is a registered trademark of The MathWorks, Inc.

[2] A recent paper by Arteaga [2013] takes advantage of symmetries in the loudspeaker array and a reformulation of the objective function to improve the convergence behavior of the optimization process.

ers regardless of spacing, hence they are louder in directions where there are more speakers. In practice, neither provides an adequate starting point for the optimization process.

The general problem is that it is difficult to pull the sound image beyond the space where there is dense coverage. For the case of hemispheres this not only means that performance will suffer below the horizon, but that it will be poor at the horizon. Because horizontal performance is uniquely important, it is necessary to make the decoder perform well there, despite the difficulties.

New design techniques have been proposed over the last few years to handle these sorts of arrays. We have implemented these in the toolbox to make them available to a wider user group. The toolbox has been extended beyond third-order decoding, and to support component order and normalization conventions other than Furse-Malham. We also wanted to support a variety of plug-in architectures. A new decoder engine was written in the FAUST (Functional Audio Stream) DSP Specification language [Orlarey, Fober, and Létz 2009; Smith 2013a], which includes facilities for dual-band decoding, and near-field, distance, and loudness compensation.

1.1 Auditory Localization

In this paper we utilize Gerzon's two main localization models to predict decoder performance: the velocity localization vector, $\mathbf{r_V}$, and the energy localization vector, $\mathbf{r_E}$. These are defined and discussed in our previous paper on the toolbox [Heller, Benjamin, and Lee 2012] (and many other places). Briefly, these models encapsulate the primary interaural time difference (ITD) and interaural level difference (ILD) theories of auditory localization. The direction of each indicates the direction of the localization perception, and the magnitude indicates the quality of the localization. In natural hearing from a single source, the magnitude of each is exactly 1 and the direction is the direction to the source.

1.2 Math Notation

We use lowercase bold roman type to denote vectors ($\mathbf{v}$), uppercase bold roman type to denote matrices ($\mathbf{M}$), italic type to denote scalars (s), and sans serif type to denote signals (W). A scalar with the same name as a vector denotes the magnitude of the vector. A vector with a circumflex ("hat") is a unit vector, so, for example, $\hat{\mathbf{r}}_\mathbf{E} = \mathbf{r_E}/r_E$. "$\mathbf{A}^\dagger$" is the Moore-Penrose pseudoinverse of $\mathbf{A}$ (`pinv(A)` in MATLAB) and "$\mathbf{A}^\top$" is the transpose of $\mathbf{A}$ (`A.'` in MATLAB).

2 Decoder Design Techniques for Domes and Multilevel Rings

In Ambisonics, the standard technique for deriving the basic decoder matrix, $\mathbf{M}$, is to invert the matrix, $\mathbf{K}$, whose columns are composed of the spherical harmonics sampled at the speaker positions, such that $\mathbf{M}\,\mathbf{K} = \mathbf{I}$, where $\mathbf{I}$ is the identity matrix [Gerzon 1980; Heller, Lee, and Benjamin 2008].[3]

Because $\mathbf{K}$ is "encoding" the speaker positions, some authors call it the *reencoding matrix* and refer to the inversion as *mode matching*. In the general case, $\mathbf{K}$ is rank deficient, so the inversion must be done by least-squares or by using singular-value decomposition (SVD) and the Moore-Penrose pseudoinverse.

Problems arise when a given loudspeaker array does a poor job of sampling some of the spherical harmonics, such as sampling at or near zero crossings or having more than one zero crossing between samples. In these cases, $\mathbf{K}$ will be ill-conditioned (difficult to invert without loss of precision) and the resulting decoder will have greater energy gain in certain directions, resulting in reduced r_E and greater loudness in those directions.

In the following subsections, we discuss three strategies implemented in the toolbox:

- Use an inversion technique suited to ill-conditioned problems
- Invert a well-behaved full-sphere coverage array, map to the real array
- Derive a new set of basis functions for which the inversion is well behaved

2.1 Modified Inversion

One proposed solution is to set all of the singular values to 1 when computing the pseudoinverse [Pomberger and Zotter 2012]. This has the effect of diminishing the use of the poorly sampled spherical harmonics. The resulting decoder has constant energy (hence, loudness) in all directions, at the expense of increased directional errors.

Another solution is to use a *truncated* SVD when computing the pseudoinverse. This simply discards the poorly sampled spherical harmonics. In the conventional pseudoinverse (e.g., as

[3] The term *sampling* is used here to mean evaluating the given spherical harmonic function at a particular azimuth and elevation.

implemented in MATLAB), normalized singular values[4] less than 10^{-15} are not inverted. In a truncated SVD, a much larger threshold is used. For example, setting the threshold to $\frac{1}{2}$ puts an upper limit of 3 dB on the loudness variations, again, at the expense of increased directional errors.

The toolbox also can produce decoders that are a linear combinations of conventional pseudoinverse and these alternatives, providing a single parameter to tradeoff uniform loudness and directional accuracy. Other approaches to inverting ill-conditioned matrices have been applied to this problem, such as Tikhonov regularization [Poletti 2005] and LASSO (least absolute shrinkage and selection operator) [Chen and Huang 2013]. Currently, we have not implemented these, although the linear combination approach described above provides a result similar to Tikhonov regularization.

2.2 Hybrid Ambisonic-VBAP Decoding

The hybrid Ambisonic-VBAP approach is called "All Round Ambisonic Decoding" (AllRAD) by Zotter and Frank [2012]. Briefly, one computes a decoder for a uniform array of virtual speakers and then maps the signals for the virtual array to the real loudspeaker array using Vector Base Amplitude Panning (VBAP) [Pulkki 1997].

VBAP always produces the smallest possible angular spread of energy for a given panning direction and speaker array, hence the perceived size of a virtual source changes depending on direction. This is directly at odds with the Ambisonic approach, which tries to keep the perceived size of a virtual source constant regardless of source direction. AllRAD uses two strategies to mitigate this:

1. The number of virtual speakers is made much larger than the number of real speakers.
2. *Imaginary speakers* are inserted to fill in large gaps in the real loudspeaker array in order to keep the triangular faces of the tessellation as regular as possible.

AllRAD places the virtual speakers according to a *spherical t-design* [Hardin and Sloane 2002]. A spherical t-design of degree t is a finite set of points on a sphere, such that the integral of any polynomial of degree t or less over the sphere is equal to the average value of the polynomial sampled at the points in the set. The present implementation uses the 240-point spherical t-design for the virtual array, which is the largest currently-known t-design.

[4] the set of singular values divided by the largest one

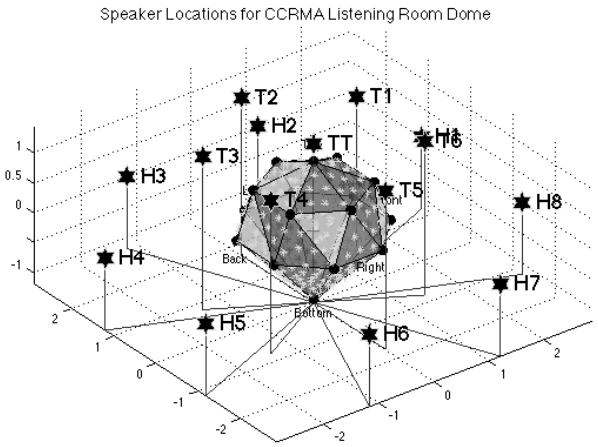
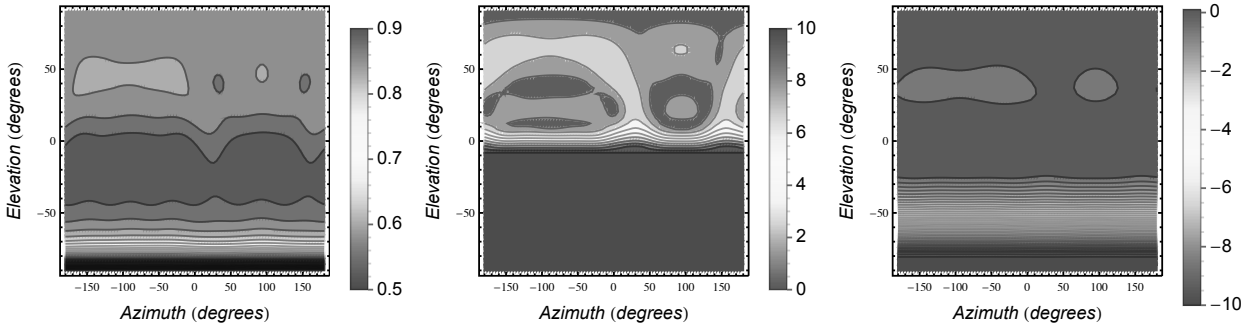


Figure 1: Plot of real speaker locations for the upper hemisphere in CCRMA's Listening Room (black hexagrams), unit sphere tessellation, and intersection points of 240 virtual speaker directions (green plus sign). The speaker at the bottom is an imaginary speaker added to keep the facets of the tessellation as regular as possible. The location of the intersection points are used to calculate the VBAP gains to the real speakers.

There are three steps to the design of an AllRAD decoder:

1. Select a spherical t-design for the array of virtual speakers and compute a decoder for it. Because the virtual speakers are distributed uniformly on the sphere the inversion is well behaved.
   (a) Compose the matrix $\mathbf{K_V}$ whose columns are the spherical harmonics sampled at the directions of the virtual speakers.
   (b) Compute the decoder matrix for the virtual array, $\mathbf{M_V} = \mathbf{K_V}^\dagger$.
2. Compute the matrix of VBAP gains for each virtual speaker.
   (a) Project the positions of the real speakers onto the unit sphere.
   (b) Add imaginary speakers to the array to fill in any gaps larger than 90°. For a dome this will be one at the bottom. For a multilevel ring, one at the top and one at the bottom. The distance from the center determines how quickly



(a) r_E vs. Test Direction (b) r_E Direction Error (degrees) (c) Energy Gain (dB)

Figure 2: The AllRAD decoder's performance for the upper hemisphere of CCRMA's Listening Room. These show the (a) energy concentration, (b) directional accuracy, and (c) loudness of sources from various directions. Directional errors are clipped at 10° so that smaller errors can be seen. The plots have been quantized to make the structure clearer. Note that the Mercator projection used overemphasizes the poles.

sources fade as they move outside the region of the sphere covered by the real speaker array.

(c) Compute the triangular tessellation of the convex hull of the projected speaker positions.

(d) Determine the intersection point of the vector to each virtual speaker with the faces of the convex hull.

(e) Calculate the barycentric coordinates of each intersection point. These are the VBAP gains from that virtual speaker to the three real speakers at the vertices of the face.

(f) Assemble the matrix of the VBAP gains, $\mathbf{G}_{\mathbf{V}\to\mathbf{R}}$. This matrix has one column for each virtual speaker and one row for each real speaker. Each column will have up to three gains for that virtual speaker from the previous step. Gains to imaginary speakers are omitted.

3. The basic decoder matrix is

$$\mathbf{M} = \mathbf{G}_{\mathbf{V}\to\mathbf{R}}\,\mathbf{M}_{\mathbf{V}}.$$

Figure 1 shows the real and imaginary speaker positions, the tessellation of the speaker directions, and the intersection points of the vectors to each virtual speaker with the faces of the tessellation. The example shown is for the upper hemisphere of loudspeakers in CCRMA's Listening Room. Figure 2 shows the performance of the AllRAD decoder used in the listening tests.

2.3 Spherical Slepian Function Decoding

Spherical Slepian functions (SSF) are linear combinations of spherical harmonics that produce new basis functions that are approximately zero outside the chosen region of the sphere, but also remain orthogonal within the region of interest. This makes them suitable for decomposing spherical-harmonic models into portions that have significant energy only in selected areas [Beggan et al. 2013; Simons, Dahlen, and Wieczorek 2006]. They have been used in satellite geodesy to model the magnetic and gravitational fields of the earth from satellite data that does not cover the whole earth. In designing Ambisonic decoders, they allow us to specify a region of interest on the sphere and derive a new set of basis functions that is well conditioned within that region. Zotter et al. call this "Energy-Preserving Ambisonic Decoding" (EPAD) [2012]. The procedure implemented in the toolbox is described here.

1. Define the subset of the surface of the sphere for the decoder, $\mathcal{R} \subset S^2$, where S^2 denotes the surface of the unit sphere in $\mathbb{R}^3$. To assure good performance at the boundary, select it to be a bit larger than the area covered by the loudspeakers; for the decoder tested, we used $-30°$ to $90°$ elevation.

2. Compose the Gramian matrix, $\mathbf{G}$, of the inner products of the real spherical harmonics, $Y_{lm}(\hat{\boldsymbol{\theta}})$, over the region $\mathcal{R}$. Each element, $g_{lm,l'm'}$, of $\mathbf{G}$ is given by

$$\begin{aligned} g_{lm,l'm'} &= \langle Y_{lm}, Y_{l'm'} \rangle_{\mathcal{R}} \\ &= \int_{\mathcal{R}} Y_{lm}(\hat{\boldsymbol{\theta}})\; Y_{l'm'}(\hat{\boldsymbol{\theta}})\; d\boldsymbol{\theta} \end{aligned}$$

where lm is a single-index designator for the real spherical harmonic of degree l and order m, $\hat{\boldsymbol{\theta}} = \begin{bmatrix}\cos\theta\cos\phi & \sin\theta\cos\phi & \sin\phi\end{bmatrix}^\top$, and θ and ϕ are azimuth and elevation.

3. Compute the eigen decomposition of $\mathbf{G} \rightarrow \mathbf{U}\,\mathbf{\Lambda}\,\mathbf{U}^{-1}$. $\mathbf{U}$ is a unitary matrix whose columns are the eigenvectors of $\mathbf{G}$. The diagonal elements of $\mathbf{\Lambda}$ are the corresponding eigenvalues.

4. Compose a new matrix, $\mathbf{U}_{\text{SSF}}$, by selecting the columns of $\mathbf{U}$ with eigenvalues above some threshold, α. α should be approximately the fraction of the sphere covered by the region of interest. For a hemispherical dome, we use $\alpha = \frac{1}{2}$. This matrix transforms points in the spherical harmonic basis to points in the new SSF basis.

5. Compose the speaker reencoding matrix, $\mathbf{K}$, where the columns are the spherical harmonics sampled at each speaker direction. Transform it to the new basis, $\mathbf{K}_{\text{SSF}} = {\mathbf{U}_{\text{SSF}}}^\top\,\mathbf{K}$

6. Compute the basic decoder matrix, $\mathbf{M} = {\mathbf{K}_{\text{SSF}}}^\dagger\,{\mathbf{U}_{\text{SSF}}}^\top$.

Figure 3 shows balloon plots of the all 16 spherical Slepian basis functions for the region $-30°$ to $90°$ elevation on the sphere. Note that the first eight are concentrated in the upper hemisphere, the next two in the middle, and the last six in the lower hemisphere. The first 13 (those with $\lambda > \frac{1}{2}$) were used for the third-order decoder we tested. One observation is that this method creates basis functions that have a clearer relationship with source directions, which is not possible for the spherical harmonics above first order. Figure 4 shows the performance of the SSF decoder used in the listening tests.

2.4 Max-r_E Decoders

The basic decoder matricies, $\mathbf{M}$, calculated in the preceding sections, are transformed into max-r_E decoders by multiplying by a matrix, $\mathbf{\Gamma}$, whose diagonal entries are the per-order gains that maximize r_E over the sphere. $\mathbf{M}_{\text{max-}r_E} = \mathbf{M}\,\mathbf{\Gamma}$. The calculation of these gains is discussed in the appendix of [Heller, Benjamin, and Lee 2012].

3 In-situ Performance Measurements

The Ambisonic decoder design philosophies discussed above are generally intended to optimize the psychoacoustically based parameters of the Gerzon Energy Vector theory. It is expected that those parameters generally predict the subjective performance of the system but, they are not the same as the parameters that directly predict what is heard by the listeners. We use measurements of the ITD and ILD to gauge the localization performance in actual systems. ITDs are known to predict localization of low-frequency sounds and ILDs are known to predict the localization of high-frequency sounds.

A group of measurements were performed in CCRMA's Listening Room at Stanford University.[5] That room is equipped with 22 loudspeakers arranged as a horizontal ring of eight loudspeakers, rings of six loudspeakers at $+40°$ and $-50°$ elevation, and one loudspeaker each at the zenith and nadir. This allowed the option of either using the full spherical array or decoders designed specifically to drive the upper 15 loudspeakers as a hemisphere. One decoder was derived by using the AllRAD method and the other by using a SSF basis set.

The ITDs and ILDs created by real systems were measured by using a dummy head to record test signals reproduced from a variety of directions. The test signals are ambisonically panned exponential sine sweeps from which the impulse response is computed from each direction. Those impulse responses are binaural impulse responses, from which the ITDs and ILDs can be derived.

The ITDs were calculated by band-pass filtering the impulse responses to the bandwidth of interest and comparing the time of arrival at the two ears of the dummy head. Performing the calculation at 192 kHz sample rate gives a time resolution of 5 µs. The measurement was repeated in each of the 37 directions at 10° intervals around the horizon, and for each of the three decoders being evaluated. The result is shown in Figure 5a. All three decoders provide a plausible ITD result. The significant differences occur at the sides.

ILDs are considerably more complex than ITDs, with the major differences between the two ears occurring at frequencies above 1 kHz. As a simplification to make comparison easier, the ILD was calculated as an average level between 1 to 4 kHz. As for the ITDs, ILD was calculated at 10° intervals around the horizon. The results are shown in Figure 5b.

The three decoders produce substantially dif-

[5] https://ccrma.stanford.edu/room-guides/listening-room

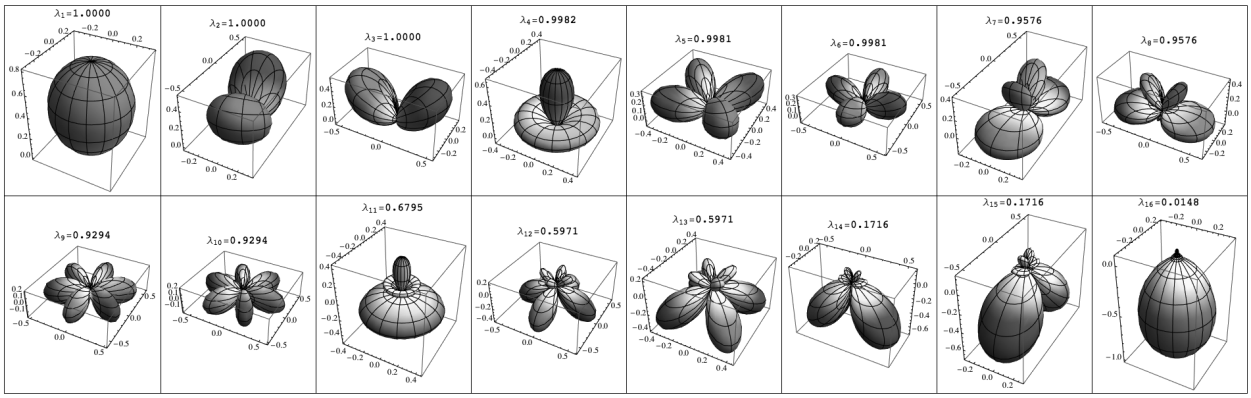
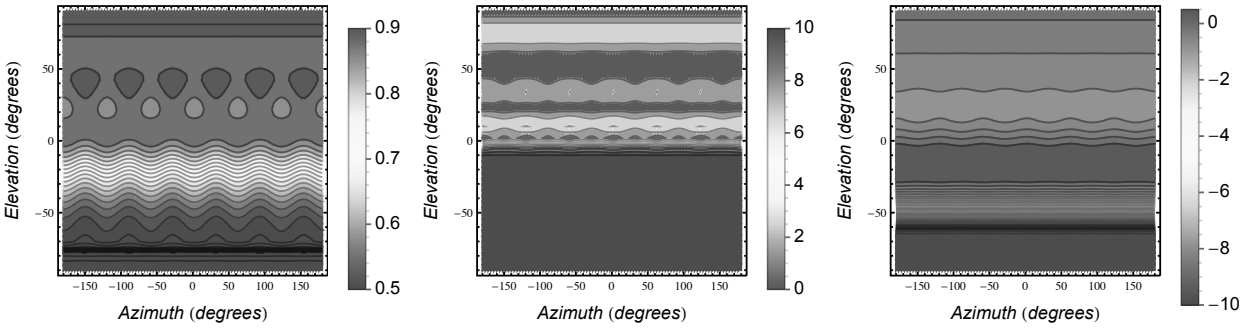


Figure 3: Balloon plots of all 16 spherical Slepian basis functions for the region $-30°$ to $90°$ elevation on the sphere. Lobes with reversed polarity are shown in blue. Note that the first eight are concentrated in the upper hemisphere, the next two in the middle, and the last six in the lower hemisphere. The first 13 ($\lambda > \frac{1}{2}$) were used for the third-order decoder we tested.



(a) r_E vs. Test Direction (b) r_E Direction Error (degrees) (c) Energy Gain (dB)

Figure 4: The Spherical Slepian function decoder's performance. These show the (a) energy concentration, (b) directional accuracy, and (c) loudness of sources from various directions. Directional errors are clipped at $10°$.

ferent values of ILD for sounds coming from the sides. It should be noted that the high values of ILD come from cancellation of signals on the opposite side of the head from the sound source by diffraction of sound traveling around the head.

Because the results of the ITD, and particularly the ILD measurements, are so complex the analysis of their effect is quite difficult and beyond the scope of the present paper. That analysis will be published in a subsequent paper.

4 Listening tests

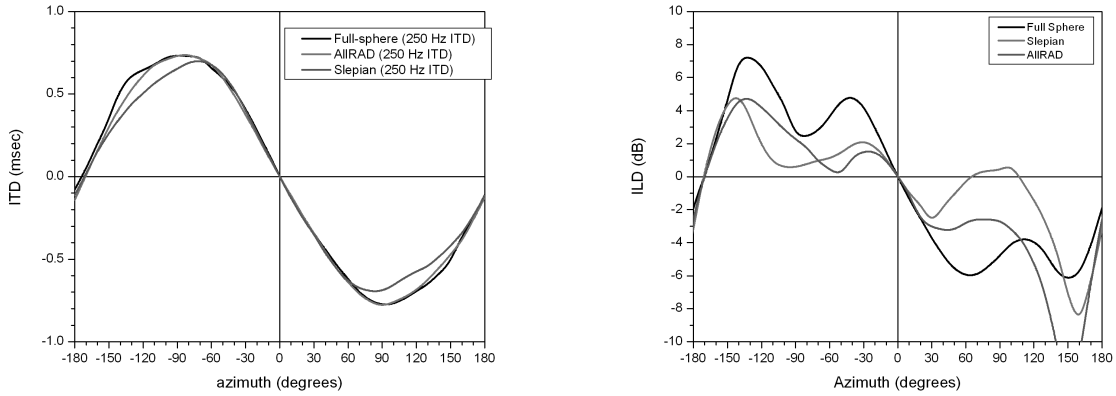
We conducted informal (non-blind) listening tests of third-order, single-band max-r_E AllRAD and SSF-based decoders using the 15 loudspeakers comprising the upper hemispherical dome in the Listening Room at Stanford's CCRMA. The decoders computed by the toolbox were saved as AmbDec configuration files and loaded into multiple instances of AmbDec so that rapid comparisons could be made.

As a reference, we also listened to full-sphere playback of the test material over all 22 loudspeakers in the Listening Room using the third-order, two-band, decoder described in the previous paper [Heller, Benjamin, and Lee 2012]. Playback levels of all three decoders were matched by ear.

The test material comprised two third-order recordings, a full-sphere mix by Jay Kadis, CCRMA's audio engineer, of "Babel" by Allette Brooks[6] and Jörn Nettingmeier's recording of *Chroma XII* by Rebecca Sanders [Nettingsmeier 2012]. Playback was directly from the Ardour sessions for each piece, which gave us the capability to move individual elements of the mix spatially to test performance from a wider variety of directions, as well as solo individual tracks.

In general, both decoders sounded quite good, providing compact and directionally accurate

[6] http://www.cdbaby.com/cd/allette4



(a) 250 Hz ITD

(b) 1 to 4 kHz ILD

Figure 5: Interaural time difference (ITD) and interaural level difference (ILD) as a function of azimuth for full-sphere, AllRAD, and SSF-based decoders. Source elevation is 0°.

imaging down to the horizontal limit of the playback array. Sources below the horizon were reproduced at the horizon, fading out as they were panned towards the nadir. The SSF-based decoder sounded brighter and more detailed than the AllRAD decoder, despite the fact that neither decoder used frequency-dependent decoding. It was also noted that with the AllRAD decoder as the listener leaned to the left and right, central sources moved in the opposite direction, whereas with the SSF-based decoder central sources remained in place.

Neither of the test decoders sounded as good as the reference dual-band, full-sphere decoder, especially in the reproduction of lower frequency percussion, which lost some of its impact. This may be attributable to the use of correct low-frequency velocity decoding ($r_V = 1$) in the reference decoder vs. wideband max-r_E decoding in the test decoders.

At the end of the listening session, we used a first-order SSF-based decoder to briefly audition a first-order Soundfield microphone recording of an orchestra made by one of the authors.[7] In this case, the instrumental balance of the orchestra was incorrect; notably, the woodwinds were almost inaudible. After the listening session, we recalled that in this recording, the microphone was hung vertically, approximately 3 meters behind and 1.5 meters above the conductor's head, placing the entire orchestra in the lower hemisphere of the recording. The first-order SSF-based decoder starts fading sources at approximately 20° *above* the horizon, which caused the instruments at the front of the orchestra to be attenuated significantly. At this point, we cannot recommend this configuration for first-order program material with significant sources in the lower-hemisphere. Possible workarounds we intend to try include inverting the vertical signal, Z, to mirror the soundfield across the $Z = 0$ plane or rotating the soundfield about the Y-axis ("tilt") in order to move important sources to the upper hemisphere.

AllRAD decoders generated by toolbox have been used for performances at Stanford's Bing Concert Hall and Studio employing CCRMA's 24-speaker, hemispherical dome, loudspeaker array. At the dress rehearsal for a performance in the Concert Hall, we were able to compare the new AllRAD decoder to the projection decoder that had been used for previous concerts. The improvement was clearly audible to all present, with increased clarity and directional focus, especially for sources behind and above the audience.

Good results have also been reported using modified inversion for a second-order decoder for a 12-speaker trirectangle array that is limited by the ceiling height of the room, leaving a large gap in coverage at the top and bottom of the array.

5 Decoding Engine

To support operation beyond third-order, a variety of plug-in architectures, and use with third-party SDKs, a new Ambisonic decoder engine

[7] Beethoven: Sym. No. 4 in B-flat Major, Op. 60, 4th Mvt. Available at http://www.ambisonia.com/Members/ajh/ambisonicfile.2008-10-30.6980317146

was implemented in FAUST. FAUST is a DSP specification language, which can target a variety of plug-in formats and operating systems.

The new implementation comprises about 250 lines of FAUST. It has no inherent limits on the Ambisonic order at which it operates and supports three modes of decoding: one decoding matrix with per-order gains ($\mathbf{\Gamma}$), one decoding matrix with phase-matched shelf filters, and dual-band, with phased-matched bandsplitting filters and two decoding matrices. The outputs can be delay and level compensated for speakers at different distances from the center of the array.

Nearfield compensation is supplied by digital state-variable realizations of Bessel filters [Smith 2013b] and can be applied at the input or output of the decoder, or turned off completely. The current implementation provides filters for operation up to fifth-order, although the toolbox includes facilities for automatically generating filters up to approximately 25th order.[8]

User adjustments are supplied for overall gain and muting, as well as crossover frequency and relative levels of high and low frequencies. All realtime controls are "dezippered" and can be accessed directly through GUI elements or via Open Sound Control.

In practice, the toolbox writes out the configuration section of the decoder and appends the implementation section, producing a single FAUST "dsp" file, containing the full decoder. The FAUST compiler (either online or local) is used to produce a highly optimized C++ class that implements the decoder, which is then wrapped in a plug-in-specific architecture file that provides the interface to the various SDKs. This is compiled to produce the plug-in file. At the time of this writing VST, AU, MaxMSP, Pd, LADSPA, LV2, Supercollider, and many others are supported on Windows, MacOSX, and Linux. In addition, an online compiler is available.

The decoder engine implementation can be used apart from the toolbox by editing the configuration options and inserting the per-order gains and matrix coefficients manually. Facilities are provided to generate configuration sections directly from existing AmbDec configuration files.

6 Channel-Order, Normalization, and Mixed-Order Conventions

At present, there are a number of channel-order and normalization conventions in use by the Ambisonics community. The toolbox implements all conventions known to the authors, including variants that adjust the gain of the omnidirectiontal component (W) to be compatible with B format. Internally, each channel is annotated with its degree, order, gain relative to full orthonormalization (N3D), and Condon-Shortly phase, so additional conventions can be added easily, if needed.

Two mixed-order conventions are supported by the toolbox: the scheme used in the AMB Ambisonic File Format (`#H#P`) [Dobson 2012] and one proposed by Travis [2009], which gives resolution-versus-elevation curves that are flatter in and near the horizontal plane (`#H#V`).

7 Conclusions and Future Work

We have reported on extensions to the Ambisonic Decoder Toolbox to handle popular loudspeaker configurations that do not cover the full sphere, such as hemispherical domes and multilevel rings. It also has been extended to operate at higher Ambisonic orders and with alternate channel order and normalization conventions. To support that, and multiple plug-in architectures, we have written a new, full-featured decoder in FAUST.

In general, the ability to generate decoders quickly has proven valuable in performance settings where one has to set up quickly and the speakers are not necessarily installed in the planned locations. The other effect is that it places less emphasis on performance prediction in that a number of decoders can be generated with different methods and parameter settings, and then auditioned to determine the best one for a particular set of playback conditions.

Generating dual-band decoders from these alternate methods is an obvious extension for the toolbox, as is using the decoders as initial estimates for the optimizer. Users have requested adding bass management to the decoder implementation. We have also investigated hosting the toolbox on a server and linking directly to the online FAUST compiler, so that a user does not need to install any software to use it.

As highlighted at the end of our listening session, a significant open question with partial-coverage decoders is what should happen if a source moves into a "poor" area, for example, the zenith or nadir directions. The effect of a Spitfire flying low overhead is probably not compromised if it appears too loud or doesn't have exact localization. Conversely, a source moving

[8] The limit is imposed by MATLAB's `roots()` function.

underground may be allowed to fade.[9]

The current implementations simply discard these sources, fading out as they are panned beyond the coverage region. In the case of the AllRAD decoders, they can be brought out for further processing by simply making the imaginary speakers into real speakers in the configuration file; however, these signals cannot be simply mixed into existing speaker feeds as the coherent combination of the signals will distort the directional fidelity of the decoder, especially for sources near the horizon. One proposal is to decorrelate them using a broadband 90° phase shift and sum into the speaker feeds. Other suggestions are welcome.

The toolbox is open source and available under the GNU Affero General Public License, version 3. The FAUST code generated by the toolbox is covered by the BSD 3-Clause License, so that it may be combined with other code without restriction. Contact the authors to obtain a copy of the toolbox.

8 Acknowledgements

The authors thank Fernando Lopez-Lezcano, who encouraged us to address this topic and helped carry out the measurements and listening tests. We also thank Andrew Kimpel, Marc Lavallée, and Paul Power who have been using the toolbox and have provided helpful feedback and discussion, and Richard Lee, Jörn Nettingsmeier, Bob Oldendorf, and the anonymous referees who made several suggestions that improved the paper.

References

Arteaga, Daniel (May 2013). "An Ambisonics Decoder for Irregular 3-D Loudspeaker Arrays," in: *134th Audio Engineering Society Convention*. Rome.

Beggan, Ciarán D. et al. (Mar. 2013). "Spectral and spatial decomposition of lithospheric magnetic field models using spherical Slepian functions," in: *Geophysical Journal International* 193.1, pp. 136–148.

Chen, Fei and Qinghua Huang (2013). "Sparsity-based higher order ambisonics reproduction via LASSO," in: *Signal and Information Processing (ChinaSIP), 2013 IEEE China Summit & International Conference on*, pp. 151–154.

[9]Since the treatment of these sources depends to some degree on the producer's intent, we suggest that new full-sphere sound transmission standards, such as MPEG-H 3D Audio, should include provisions for "rendering hints", along the lines of the downmix metadata in Dolby Digital. [Dolby Laboratories, Inc 2005]

Dobson, Richard (2012). *The AMB Ambisonic File Format*. Accessed 1 Feb 2014. URL: http://people.bath.ac.uk/masrwd/bformat.html.

Dolby Laboratories, Inc (Oct. 2005). *Dolby Metadata Guide*. Tech. rep. S05/14660/16797.

Gerzon, Michael A. (Feb. 1980). "Practical Periphony: The Reproduction of Full-Sphere Sound," in: *65th Audio Engineering Society Convention Preprints*. 1571. London.

Hardin, R. H. and N. J. A. Sloane (2002). *Spherical Designs*. Accessed 1 Feb 2014. URL: http://neilsloane.com/sphdesigns/.

Heller, Aaron J., Eric M. Benjamin, and Richard Lee (Mar. 2012). "A Toolkit for the Design of Ambisonic Decoders," in: *Linux Audio Conference 2012*, pp. 1–12.

Heller, Aaron J., Richard Lee, and Eric M. Benjamin (Dec. 2008). "Is My Decoder Ambisonic?" In: *125th Audio Engineering Society Convention, San Francisco*, pp. 1–21.

Nettingsmeier, Jörn (Mar. 2012). "Field Report II – Capturing *Chroma XII* by Rebecca Saunders," in: *Linux Audio Conference 2012*.

Orlarey, Yann, Dominique Fober, and Stephane Létz (2009). "Faust: an Efficient Functional Approach to DSP Programming," in: *New Computational Paradigms for Computer Music*. Ed. by Gérard Assayag and Andrew Gerzso. Delatour.

Poletti, Mark (Nov. 2005). "Three-Dimensional Surround Sound Systems Based on Spherical Harmonics," in: *Journal Of The Audio Engineering Society* 53.11, p. 1004.

Pomberger, Hannes and Franz Zotter (Mar. 2012). "Ambisonic panning with constant energy constraint," in: *DAGA 2012, 38th German Annual Conference on Acoustics*.

Pulkki, Ville (June 1997). "Virtual Sound Source Positioning Using Vector Base Amplitude Panning," in: *Journal Of The Audio Engineering Society* 45.6, pp. 456–466.

Simons, Frederik J., F. A. Dahlen, and Mark A. Wieczorek (Sept. 2006). "Spatiospectral Concentration on a Sphere," in: *SIAM review* 48.3, pp. 504–536.

Smith, Julius O (June 2013a). *Audio Signal Processing in FAUST*. Accessed 1 Feb 2014. URL: https://ccrma.stanford.edu/~jos/aspf/.

Smith, Julius O. (2013b). *Digital State-Variable Filters*. Accessed 1 Feb 2014. URL: https://ccrma.stanford.edu/~jos/svf.

Travis, Chris (June 2009). "A New Mixed-Order Scheme for Ambisonic Signals," in: *Proc. 1st Ambisonics Symposium*, pp. 1–6.

Zotter, Franz and Matthias Frank (Nov. 2012). "All-Round Ambisonic Panning and Decoding," in: *Journal Of The Audio Engineering Society* 60.10, pp. 807–820.

Zotter, Franz, Hannes Pomberger, and Markus Noisternig (Jan. 2012). "Energy-Preserving Ambisonic Decoding," in: *Acta Acustica united with Acustica* 98.1, pp. 37–47.

WiLMA
a *Wi*reless *L*arge-scale *M*icrophone *A*rray

Christian SCHÖRKHUBER,
Markus ZAUNSCHIRM and
IOhannes m zmölnig
Institute of Electronic Music and Acoustics (IEM)
University of Music and Performing Arts, Graz, Austria
{schoerkhuber, zaunschirm, zmoelnig}@iem.at
http://wilma.iem.at

Abstract

Everyday situations are rich in numerous acoustic events emerging from different origins. Such acoustic scenes may comprise discussions of our fellow human beings, chirping birds, cars, cyclists, and many more. So far, no recording or scene analysis technique for this rich and dynamically changing acoustic environment exists, though it would be needed in order to document or actively shape an acoustic scene. We know customised techniques for recording symphony orchestras with a static cast, but none that automatically readjusts to scenes with varying content. Thus, a new recording technique that analyses the signal content, the position and the activity of all sources in a scene, is required. We present *WiLMA*, a wireless large scale microphone array, a mobile infrastructure that allows for investigating into new recording and analysis techniques.

Keywords

network audio, sensor network, microphone, distributed processing

1 Introduction

Traditionally, the sensor nodes of a wireless sensor network (WSN) that captures sound events, are populated with low quality microphones, amplifiers and analogue to digital converters (ADCs) in order to decrease sensor node size, power consumption and cost.

The Wireless large-scale microphone array (WiLMA) introduces high quality audio processing in wireless sensor networks. Each of the sixteen sensor modules (SM) allows for capturing of up to four high-end microphone signals which in turn enables the use of a 4-channel microphone array (e.g. first order tetrahedral ambisonics microphone) per SM. Thus, the system operates as a large scale microphone array, with a total of 64 audio channels. A single SM and the used microphone array are depicted in fig.1.

The acquired data from all SMs is transmitted (either wireless or wired) to a central unit (CU) running the host application shown in fig.6 and fig.7. This host application visualises input levels, synchronisation and battery status. Further, it allows the user to individually configure each SM for a specific task.

Figure 1: Sensor module and microphone array (*Oktava 4D-ambient*)

Each SM is equipped with a local processing unit in order to perform computations on the acquired data. Instead of sending the raw data to the the central unit responsible for the fusion, sensor modules can use their processing abilities to locally carry out simple computations and transmit only the required and partially processed data. This intelligent sensor network approach results in decreased network traffic and higher flexibility of the system.

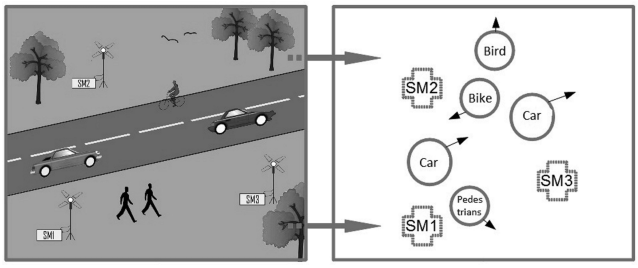


Figure 2: Acoustic scene analysis

An example application using the WiLMA hardware is to separate sources of an acoustic scene and track their movement. Thus, it should be possible to analyse the separated source signals and to assign a specific event to

a specific source. Fig.2 conceptually depicts the process of a spatial transcription. Areas that could benefit from that application include assisted living scenarios, acoustical planning, the surveillance of urban areas, multichannel source separation, event detection, source tracking and so on. Another application is the high audio quality multichannel recording of an acoustic scene with the added benefit of flexible microphone positioning due to wireless operation of the system.

2 Design

The basic design of the sensor network contains a *Central Unit* and a variable number of *Sensor Modules*.

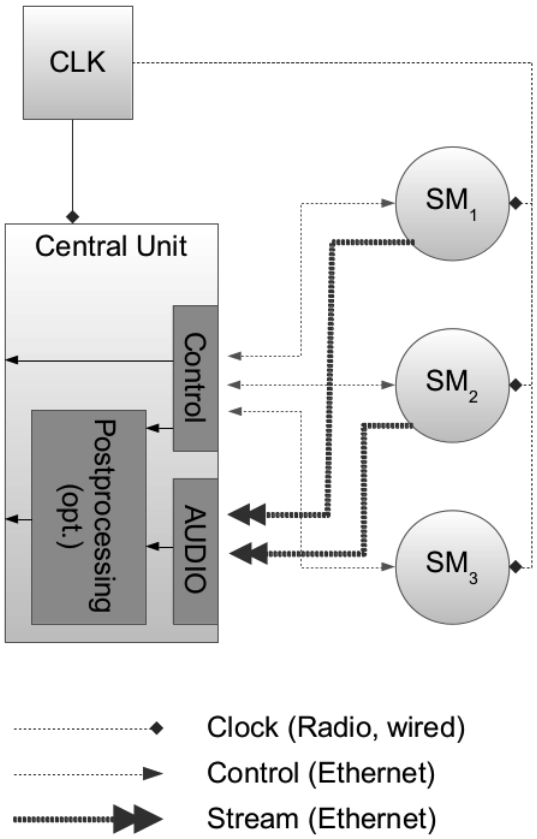


Figure 3: Network diagram of multiple synced Sensor Modules and a Central Unit

The Central Unit controls and monitors the individual modules, The Sensor Modules capture audio autonomously and send their data to the Central Unit, where it can be collected for further processing.

To allow for sample synchronous audio capturing, all SMs are connected to a central master clock.

2.1 Modes of Operation

We can distinguish between three different modes of operation for each sensor unit:

2.1.1 Recording

The simplest operational mode is to *record* the microphone signals locally on the SM. The recording should be time-stamped, so the recording of multiple SMs can be time-aligned later in an offline process.

2.1.2 Streaming

For recording and monitoring purposes, it might often be desirable to not collect the audio data decentralised on the SMs and collect them later, but rather have all audio channels available immediately at the Central Unit, by means of real-time streaming. This allows the sensor network to be used as a de-centralised capture-only multichannel sound card.

2.1.3 Processing

Each SM is also equipped with a local processing unit that can be used to do (simple) analysis of the local signals, parallelising the computational load.

The actual processing algorithm might change depending on the application. It is therefore required to be able to implement algorithms in a reasonable environment and deploy these programs easily on all (or selected) SMs.

The result could be either an enhanced signal, meta-data about the signal or a mixture of both (e.g. using signal identification on the 4 channel recording, it is possible to only stream a mono-version of the signal together with positional meta data).

2.1.4 Mixed

Multiple connected SMs need not operate in the same mode. For instance, some SMs could be streaming audio, whereas other SMs would only do processing and send meta-data to the Central Unit (as depicted in Fig.3).

2.2 Communication

All control communication between the CU and the SMs is based on a bi-directional *OSC*-connection. Typical OSC-applications use UDP as transport protocol, which behaves badly in congested networks. In order to work around reliability issues, the transport layer can be configured to either use UDP or TCP/IP with

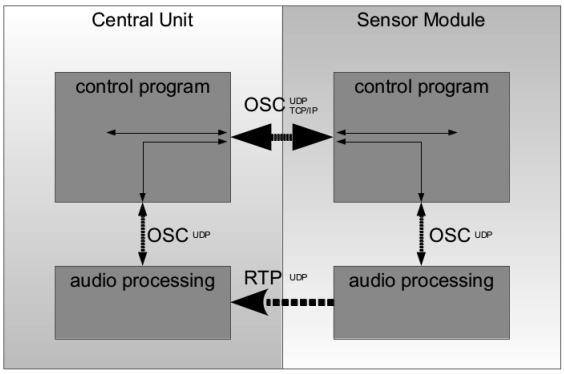


Figure 4: Communication between central unit and sensor modules

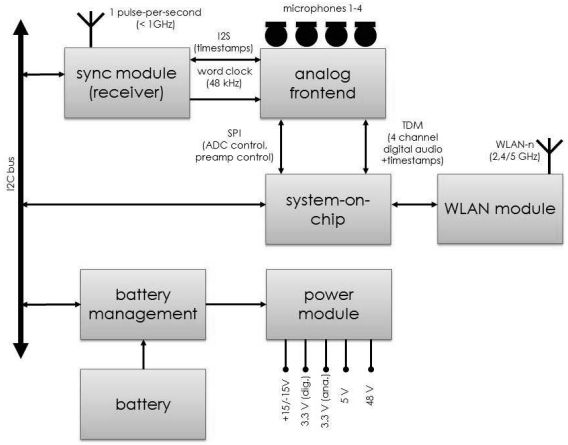


Figure 5: Block diagram of the sensor module

SLIP-based packetizing as suggested by the OSC-1.1 specifications [1].

Besides configuring and activating the various modes of operation, the "control channel" includes basic infrastructure (like sending and receiving heartbeats in order to determine whether the connection is still established (in the case of UDP) and the SM is still responsive) and health information (e.g. CPU load, memory and disk usage, battery status, sync status, microphone levels). It also allows to configure the SM (e.g. setting the gain of the microphone preamplifier) and transports the entire meta-information extracted by any optional *process*ing on the SM.

Audio streaming from the SM towards the CU is not done via OSC (as suggested e.g. by [2]), but instead uses the more widespread *RTP* protocol [3] on top of UDP. The RTP-timestamps are synchronised, in order to be able to re-align the audio signals of multiple SMs.

3 Sensor Module

The Sensor Module (see Fig.1) consists of a custom hardware design running Linux.

3.1 Audio

The 4 channel analogue front end is equipped with THAT1570 low noise, differential microphone preamplifiers which are digitally controlled via SPI using THAT5173 controller ICs. Analogue to digital conversion is performed by an AD1974, a 4 channel, 24 bit ADC with integrated phase-locked loop (PLL).

3.2 Synchronisation

The internal sampling clock of the AD1974 is derived from the word clock provided by the synchronisation module. Wireless synchronisation within the WiLMA system is established via a 1 pulse-per-second timestamp signal that is broadcasted by the master module on a sub-GHz ISM band. The synchronisation module is populated with a voltage controlled oscillator (VCXO) that is disciplined by a frequency locked loop (FLL) and a subsequent frequency divider to obtain the 48 kHz word clock for the ADC. The sample accurate timestamps generated by the synchronisation module is multiplexed with the output data of the ADC into a 8-channel/32 bit time-division multiplexing (TDM) stream.

3.3 System On Chip

The heart of each sensor module is a *Beaglebone A6* equipped with an *ARM Cortex A8* based processor running Linux. The TDM audio stream is read by an ALSA driver that sets up the ADC, controls the microphone preamplifiers and accesses the Multichannel Audio Serial Port (McASP) via the *DaVinci ASoC* driver.

3.4 Power Supply

The power module generates supply voltages for the different modules from the wall plug supply or the battery, respectively. It also generates an optional 48V supply voltage for microphones requiring phantom power. The *LiPo* battery pack is connected to a battery management system which is responsible for controlling charge voltage and charge current, switching between power sources and providing information about the battery status via I2C bus. In case of battery undervoltage the battery management system autonomously disconnects the load from

the battery to keep the battery in a safe state.

3.5 Software

Each SM is running on *Ubuntu-11.10 (Oneiric Ocelot)*, using the standard *armel* architecture packages, with the notable exception of the kernel, which is a customised build of *linux-3.2.30* due to the required ALSA drivers of the custom sound card.

When the system starts up, a control program – the *WILMAsm* daemon – is started. This daemon monitors the various health states of the system and runs an OSC-server for communication with the CU. The service is announced via ZeroConf/Avahi [4], using the type specifier `_wilma-sm._udp` (resp. `_wilma-sm._tcp`).

Since the daemon is implemented in *Python*, a more appropriate sub-system for running the audio-related tasks is needed. This subsystem has been implemented using *Pure Data*, as it is a well known environment and allows to deploy algorithm implementations in a text-based form (thus reducing the need to cross-compile binaries for the target *ARM* platform).

In order to integrate nicely with the framework, any *process*ing unit needs to adhere a simple standard, which defines inlets/outlets of the *Pd*-patch and the filesystem layout.

The used implementation of *Pd* is a slightly modified version of `Pd-0.44-2`. The main modification has been a customisation towards the special audio layout of the SM, which provides an eight channel audio interface, where only the first four channels contain actual audio data (as sampled from the microphones), and the remaining four channels contain a 32bit timestamp synchronised on all SMs.[1]

Pd is running as a sub-process of the control-daemon, which monitors the audio process and restarts it in the unlikely event of a crash. The control daemon and the audio process communicate via a bi-directional OSC connection on top of UDP. (No TCP/IP option is given here, as the connection is only running on `localhost`).

4 Central Unit

The Central Unit is an off-the-shelf Linux system eventually equipped with a MADI audio interface (in order to play back the independent audio streams from 16 SMs), and is running the audio stream aggregator and control application *WILMix*.

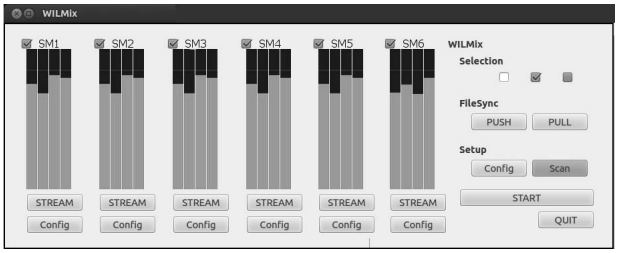


Figure 6: *WILMix* overview over available SMs

The control application provides a user-interface for controlling and monitoring the various aspects of the SMs, like starting audio streaming, distributing *process*-patches or collecting recordings.

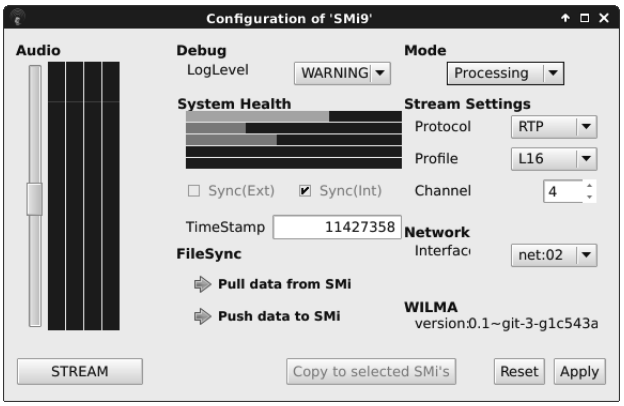


Figure 7: *WILMix* controlling a specific SM

The application uses ZeroConf to detect all available SMs in the local network, and constructs a mixer application for the given number of channels.

The audio stream aggregator receives the RTP-streams from the various SMs, and realigns them in time, so that they can be played back sample synchronously.

As is with the SMs, the control part of the application is implemented in *Python*, whereas the audio processing part is written in *Pd*, both communicating via OSC over UDP.

5 Discussion

While the current software implementation works as a proof of concept, there are certainly things to improve.

For one thing, the use of *Pure Data* on an *ARM Cortex A8* is suboptimal, as the processor

[1] Obviously this makes the timestamp encoded in a highly redundant way. The main reason for this redundancy is that the AD1974 allows to easily copy a single 32bit auxiliary digital data word into four channels at once. Since the channels 5 to 8 are unused anyhow, no immediate drawback arises from this redundant data handling.

lacks an FPU, whereas *Pd* does all processing on floating point samples.

Implementations using alternative frameworks that would allow for fix-point arithmetic (such as *GStreamer*[5]) were initially planned but were soon discarded in order to avoid cross-compilation environments altogether. (A major issue when the potential algorithm implementers are matlab-spoilt, C-agnostic students).

Even with *Pd* as the audio engine, it might be advisable to use it's library incarnation *libpd*[6] rather than a full-fledged *Pd*, as it would greatly simply the communication between the control application and the audio engine. Using *libpd*, it should even be possible to get rid of the modifications currently needed to obtain the 32bit timestamps from the audio channels[2].

6 Availability

The source code for the WiLMA-Application (running on both the SMs and the CU) has been released under the *GNU GPL*, and is available for download from *github*[3].

The hardware has beed designed in-house at the *Institute of Electronic Music and Acoustics*. However, the schematics have not yet been published under an open license.

7 Conclusions

The WiLMA hardware introduces high quality audio processing in wireless sensor networks. The overall system comprises 16 sensor modules that allow for recording up to 64 audio channels. Audio signals in the frequency range between 20Hz and 20kHz are converted with a high quality ADC (24bit). The information of each sensing module is collected by a central unit, that combines the individual data to a final outcome. Data transmission between the SMs and a central unit can either be wireless (WLAN) or wired (Ethernet). The capsules of the used microphone arrays (*Oktava 4D*) obey a linear frequency response (no sound colouration) and a minimal gain mismatch between capsules. Furthermore, the system offers a runtime of up to 8 hours in battery-powered mode. Thus, its mobile and flexible use is ensured.

In order to allow for the application of algorithms of the acoustic field theory, the audio streams of different SMs are synchronised with an accuracy of one sample ($\approx 20\mu$s).

[2]The timestamps cannot be read directly in patchspace, as *Pd* does not provide a 32bit integer type – all numbers are equal... and they are (single precision!) floats.

[3]`https://github.com/iem-projects/WILMAmix/`

8 Acknowledgements

This project was partly funded by the *MINT/Masse* program of the Austrian Federal Ministry of Science and Research.

References

[1] A. Freed and A. Schmeder, "Features and future of open sound control version 1.1 for nime," in *NIME'09: Proceedings of the 9th Conference on New Interfaces for Musical Expression*, 2009.

[2] W. Jäger, "Audio over internet using OSC," Institute of Electronic Music and Acoustics, Tech. Rep., 2010.

[3] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A transport protocol for real-time applications," *IETF RFC3550*, 2003.

[4] S. Cheshire, "Zero configuration networking (Zeroconf)," in *http://www.zeroconf.org/, accessed 2014-02-02.*

[5] GStreamer Team, "GStreamer: open source multimedia framework," in *http://gstreamer.freedesktop.org/, accessed 2014-02-02.*

[6] P. Brinkmann, P. Kirn, R. Lawler, C. McCormick, M. Roth, and H.-C. Steiner, "Embedding pure data with libpd," in *Proceedings of the Pure Data Convention*, 2011.

Extending the Faust VST Architecture with Polyphony, Portamento and Pitch Bend

Yan Michalevsky
Department of Electrical Engineering,
Stanford University
yanm2@stanford.edu

Julius O. Smith
Center for Computer Research in Music and Acoustics (CCRMA),
Stanford University
AES Fellow
jos@ccrma.stanford.edu

Andrew Best
Blamsoft, Inc.
andrew@blamsoft.com

Abstract

We introduce the `vsti-poly.cpp` architecture for the Faust programming language. It provides several features that are important for practical use of FAUST-generated VSTi synthesizers. We focus on the VST architecture as one that has been used traditionally and is supported by many popular tools, and add several important features: polyphony, note history and pitch-bend support. These features take FAUST-generated VST instruments a step forward in terms of generating plugins that could be used in Digital Audio Workstations (DAW) for real-world music production.

Keywords

Faust, VST, Plugin, DAW

1 Introduction

FAUST [5] is a popular music/audio signal processing language developed by Yann Orlarey et al. at GRAME,[1] with contributions from a community of developers. The FAUST toolset enables the generation of standalone synthesizers as well as plugins for various operating systems and environments. Considering FAUST a convenient tool and a fast way for prototyping and even creating production level sound effects and synthesizers, we would like to use FAUST in combination with real-world music production tools and DAWs (Digital Audio Workstations).

We believe it is necessary to facilitate working with tools such as Cubase, Ableton or other DAWs providing a similar level of user experience and features. In the past ten years those tools shifted from relying on built-in PC soundblaster or external MIDI-controlled modules to a plugin based architecture. Plugins are used to generate sound and apply audio effects. Several common plugin architectures exist: VST, Apple's Audio Unit (AU), LV2 (the successor of LADSPA and DSSI under Linux OS). The VST (Virtual Studio Technology) plugin standard was released by Steinberg GmbH (famous for Cubase and other music and sound production products) in 1996, and was followed by the widespread version 2.0 in 1999 [8]. It is a particularly common format supported by many older and newer tools.

Some of the features expected from a VST plugin can be found in the VST SDK code.[2] Examining the list of MIDI events [1] can also hint at what capabilities are expected to be implemented by instrument plugins. We also draw from our experience with MIDI instruments and commercial VST plugins in order to formulate sound feature requirements.

In order for FAUST to be a practical tool for generating such plugins, it should support most of the features expected, such as the following:

- Responding to MIDI keyboard events
- Polyphony
- Portamento
- Pitch-bending (wheel controlled)
- Arpeggio
- Other effects dependent on note occurrence history

All of the plugin formats mentioned above can be generated from FAUST code with varying levels of feature support. For example, there is a very complete `faust2lv2` shell-script distributed with FAUST provided by Albert Gräf [3]. There is also a highly useful `faust2au` script by Reza Payami that is still under development. Useful VST 2.4 plugins can be generated using the `faust2vst` script, and relatively limited VSTi plugins (i.e., VST synthesizer or "instrument" plugins) can be generated using `faust2vsti`. Initial VSTi support was limited

[1] `http://faust.grame.fr`

[2] Specifically in the PlugCanDos namespace, declared in audioeffectx.cpp (in VST 2.4 SDK)

a single voice (implemented in the FAUST architecture file `vsti-mono.cpp`).

This paper describes the VSTi support implemented in the FAUST architecture file `vsti-poly.cpp`.[3] This effort adds polyphony support, pitch-bend, note-history, and other features described below. Pitch-bend and note history support facilitates effects such as portamento slide,[4] and creating arpeggiators. Finally, we provide an example of how it can be used to create instruments. We demonstrate using FAUST-generated VST plugins with MuLab [4] and Renoise [7] workstations. We also discuss possible future improvements and additions.

Related work

For handling MIDI events and polyphony support in a FAUST architecture file, we benefited from the MIDI plugin section of [3] and the FAUST DSSI architecture-file source code `dssi.cpp`. Additionally, `vsti-mono.cpp` was useful as a basis for our extended FAUST VSTi architecture.

2 Design

Following the convention introduced by Albert Gräf for `faust2pd` [2] and `faust2lv2` [3] et al. [6], the VST architecture file implements functionality for recognizing the "freq", "gate" and "gain" FAUST-control labels to set the note and velocity upon MIDI Note-On events (0x90) and to set the gate to 0 for a MIDI Note-Off event (0x80). One approach to implementing polyphony for the VSTi architecture is doing it similarly to the DSSI plugin architecture. The "freq", "gate" and "gain" are mapped to the controls multiple times which enables playing simultaneously a predefined maximum number of notes.

We combine the approaches taken in `vsti-mono.cpp` and `dssi.cpp`. Figure 1 shows a UML diagram describing our design (`vsti-poly.cpp`). A VST host interacts with the VST plugin through the *AudioEffectX* interface. The `Faust` class defines the functionality of the plugin by implementing that interface. The `mydsp` class performs the signal processing and synthesis—it is the code that is actually produced by the FAUST compiler. We instantiate `mydsp` for each voice (`Voice` class).

[3] It is expected that this name will later change to `vsti.cpp`. The `faust2vsti` command-line script will of course be updated as well in that case.

[4] Although for a monophonic synthesizer portamento can be implemented by smoothing the input frequency.

The VST plugin controls are created and updated using the `vstUI` class. There is an instance of `vstUI` held by the `Faust` class which is used for knobs and sliders controlled by the user via the graphical interface or by mapping MIDI controls. This instance is for controlling parameters that are global and should affect every note played. The instances of `vstUI` that are created as part of each `Voice` instance are for controlling per note parameters (frequency, gain, previously played frequency and gate). The `Faust` class implementation of the `setParameter` interface method is broadcasting any change in the global plugin parameter to all `Voice` instances.

Handling MIDI events

FAUST VSTi architecture handles MIDI events delegated by the VST host. The host sends the events to the plugin by calling `processEvents`. An event of type `kVstMidiType` indicates a MIDI event.

Note On

A MIDI note-on event (status byte is 0x9) results in searching for a free voice instance to handle the new note in the `freeVoices` list contained in the `Faust` class. The search proceeds in a classic round robin pattern as found in hardware synthesizers. If a free voice is found, the voice is designated as the new voice, otherwise the oldest playing voice is stolen and designated as the new voice. Its frequency is set according to the note number, the gain parameter is set according to the note velocity, and the gate is set to 1. An entry is added to `playingVoices`, mapping the note to the voice index, and the voice index is removed from the `freeVoices` list. The previously played note is saved in order to enable the portamento slide.

The VST format operates with multiple samples in a processing block. The note-on event includes a sample offset within the current block. These deltas are stored in a list so that multiple note-on events can be handled in the block. The note to voice allocation occurs within the processing loop, so that each note starts at its correct sample position within the block.

Note Off

A MIDI note-off event (status byte is 0x8) results in searching for the corresponding `Voice` instance in the `playingVoices` list contained in the `Faust` class. The gate is then set to 0. Because the voice may have a release tail after the gate is zeroed, a silence detection algorithm is used to determine when the voice index should

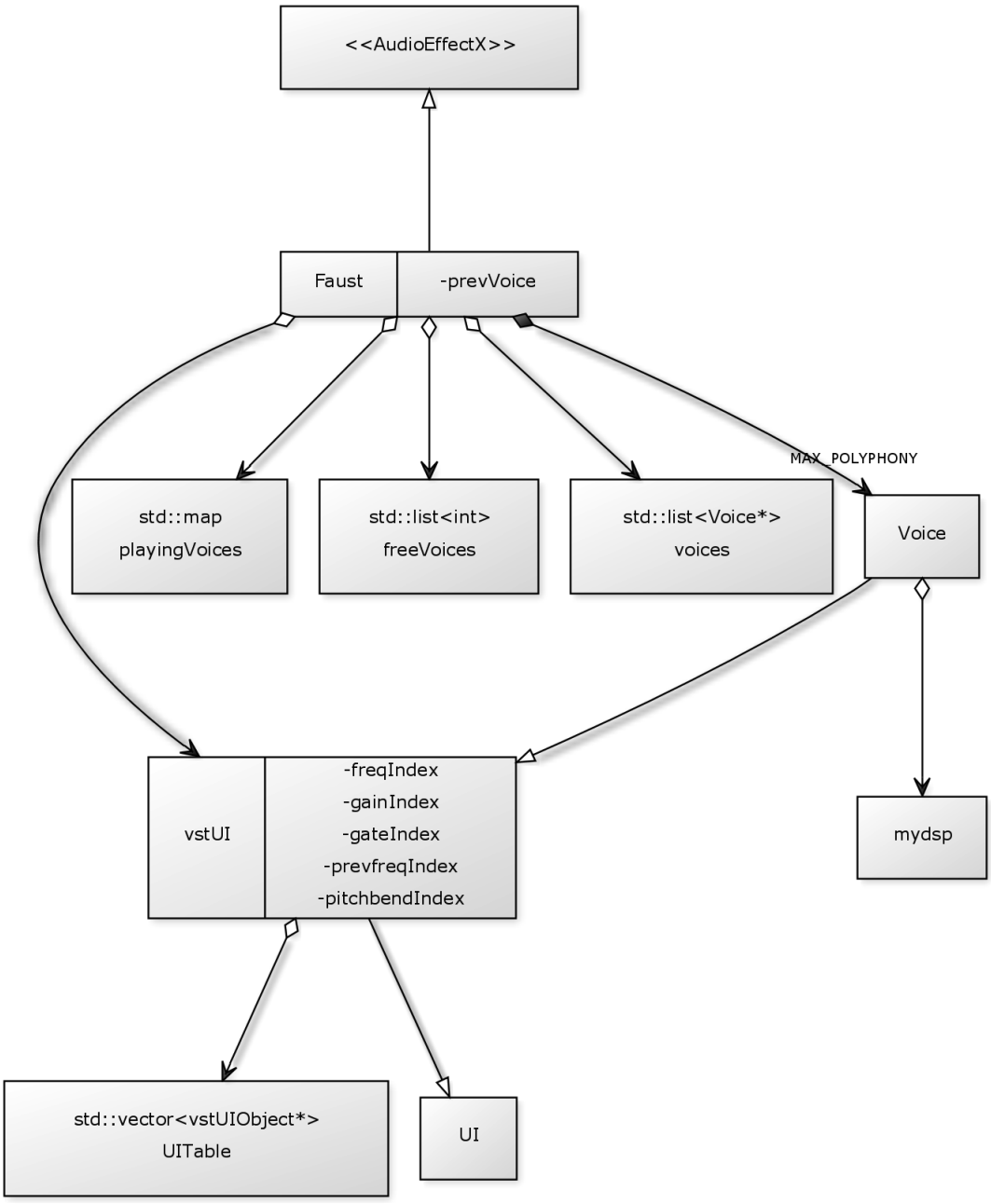


Figure 1: FAUST VSTi design

be added to the `freeVoices` list. The voice output must be below the silence threshold for an entire block before it is marked as free. Silence detection allows sounding voices to not be reallocated prematurely and also provides better CPU efficiency compared to always processing all voices. Like note-on events, note-off events are sample accurate within a block.

Pitch Bend

A MIDI pitch bend is indicated by status byte 0xE. The MIDI event pitch argument has values in the range 0..16384. We normalize it to be in the range -1..1 and broadcast the value to all voices thus affecting all currently playing notes. The frequency is not updated by the architecture, as it is the responsibility of the FAUST code to use the `pitchbend` control value. This separation enables the user to ignore or handle the pitch-bend MIDI event according to the desired behavior.

All-notes-off Event

The All-notes-off MIDI event is indicated by a note number of 0, and velocity 0. Like the single note-off event, the voice gate is set to 0 and

entered into the release silence detection state. This is done for all active voices.

Portamento Slide Implementation

We demonstrated the very common portamento slide effect by creating a FAUST VSTi based on the sawtooth synthesizer that is part of the FAUST oscillator library (`oscillator.lib`). We added a portamento control that can take values in the range 0.01..0.3. The portamento effect is achieved by mixing two exponentials, one decaying and one reaching saturation with characteristic time τ that is equal to the value of the portamento control.

$$f_{mixed} = f_{new} \cdot \left(1 - e^{-\frac{t}{\tau \cdot SR}}\right) + f_{prev} \cdot e^{-\frac{t}{\tau \cdot SR}}$$

where SR is the sampling frequency, t is the time that has passed since the new note was played and f_{new} and f_{prev} are the new and previously played frequencies, respectively. This instrument also supports pitch bending controlled by the pitch-bend wheel. The f_{new} is actually a sum of note frequency and the value of the pitch-bend control (in the range -1..1) multiplied by 20. The demo synthesizer source code is presented in Alg. 1.

A demonstration of music production using FAUST can be found at `http://stanford.edu/~yanm2/music/faustloop.mp3`.

This short loop was produced using only FAUST-generated VSTi plugins, with the exception of the drums.

Figure 2: VST plugin generated by FAUST as it appears in MuLab. Using predefined control names "freq", "gain", "gate", "prevfreq" and "pitchbend" automatically maps the controls to MIDI event parameters.

3 Installation and Basic Usage

Basic installation instructions are provided in [3]. If you are using an up-to-date version of FAUST you should already have the `vsti-poly.cpp` architecture file, and running

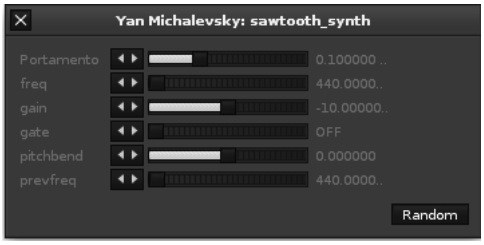


Figure 3: VST plugin generated by FAUST as it appears in Renoise tracker.

```
make install
```

should make `faust2vsti` tool accessible from any directory. Running

```
faust2vsti <yourfaustcode.dsp>
```

will create a VST effect or synthesizer from the .dsp file. To produce only the source code (.cpp) run

```
faust -a vsti-poly.cpp
-o <output filename>
<yourfaustcode.dsp>
```

`vsti-poly.cpp` currently supports both VST audio processing plugins and VSTi-MIDI-driven software synthesizer plugins. In the future we expect to consolidate all the VST related architecture files under the FAUST project.

4 Future Work

In this section we briefly offer suggestions for future development, based on our observations during this project.

Inherent portamento slide support

Portamento-slide is common to many synthesizers, for which reason it may be a good idea to incorporate the support for it into the architecture file. This effect requires a gradual change of frequency that can be performed by `vsti-poly.cpp`. The speed of transition to the new frequency could be determined by a "portamento" control as is done with other controls recognized by the architecture.

Inherent pitch-bend support

Pitch-bending is also common to many synthesizers and requires a change of frequency. This change in frequency can be done by the architecture prior to calling `mydsp::compute`[5].

[5]This of course requires the synth to use a "freq" control and not only note identifier as we suggest in the next paragraph. It would also require a way to specify the bending range.

Algorithm 1 sawtooth-synth: sawtooth with portamento and pitch-bend in FAUST

```
declare name "Sawtooth-Synth";

import("music.lib");
import("oscillator.lib");

gate = button("gate");
gain = hslider("gain[unit:dB][style:knob]", -10, -30, +10, 0.1) : db2linear : smooth(0.999);
freq = nentry("freq[unit:Hz]", 440, 20, 20000, 1);
prevfreq = nentry("prevfreq[unit:Hz]", 440, 20, 20000, 1);
portamento = vslider("[5] Portamento [unit:sec] [style:knob] [tooltip: Portamento (frequency-glide)
time-constant in seconds]", 0.1,0.01,0.3,0.001);
pitchbend = vslider("pitchbend", 0, -1, 1, 0.01);

start_time = latch(freq == freq', time);
dt = time - start_time;
expo(tau) = exp(0-dt/(tau*SR));
mix(tau, f, pf) = f*(1 - expo(tau)) + pf*expo(tau);
bended_freq = freq + pitchbend * 20;
sfreq = mix(portamento, bended_freq, prevfreq) : min(20000) : max(20);

x = sawtooth(sfreq : smooth(0.999));
process = x * gain * (gate);
```

Setting note identifier control in addition to frequency

Currently the pitch is set by a "freq" control, used by the FAUST code to determine the frequency. The "freq" control value is set by the architecture according to the note identifier received in the MIDI Note-On event. Sometimes it is more useful to have the note identifier or piano key identifier. For instance, there are existing FAUST synthesizers that take the key as input. A percussion synthesizer that produces a different sound for every key would possibly use a key identifier instead of note frequency. It would be therefore a welcome addition to the vsti-poly architecture to set the value of a note identifier control on each Note-On event.

Extended note history

We currently save only the previously played frequency enabling the implementation of the portamento-slide. Synthesizers that produce chords or arpeggios may require information about more previously played notes. This would be enabled by extending the saved note history. Passing these values to the FAUST code would require instantiation of multiple note or frequency controls.

Single FAUST VST architecture file

Currently there are several FAUST architecture files related to VST: `vst.cpp, vst2p4.cpp, vsti-mono.cpp and vsti-poly.cpp`. While theses have been kept side-by-side to not interfere with other users during development of each new architecture file, they are redundant and should be consolidated into a single `vsti.cpp` architecture file.

Shared signals among multiple voices

Many synthesizers offer modulation sources that affect multiple voices simultaneously. For example, an LFO can modulate pitch or waveform on all voices in a polyphonic synth. In the future it would be beneficial if shared signal support was provided to the synthesizer designer.

Enhanced GUI support

Other architectures within the FAUST ecosystem have more features in their GUI layout capabilities. The grouping of controls into subsections and providing specification of knobs vs. sliders would provide better flexibility and organization comparable to hand coded VSTi plugins.

Further Host-Plugin integration

One simple yet useful feature to implement is the `Bypass` capability, enabling the user to turn off a plugin from the host.

More information provided to the plugin by the host includes time and tempo. This can be useful for implementing arpeggio instruments, or audio effects dependent on tempo, such as

gating or synchronized echo.

Consolidation of various VST related architecture files

At the time of writing, FAUST code contains multiple architecture variants pertaining to VST: `vst.cpp` and `vst2p4.cpp` for effects, `vsti-mono.cpp` for monophonic instruments and `vsti-poly.cpp`, introduced by this work, supporting effects, polyphonic instruments and other features discussed in the paper. We suggest there should be one architecture encorporating all mentioned functionality. Meanwhile, since portamento is very relevant to monophonic instruments, we added the necessary modifications to support the effect in `vsti-mono.cpp`, as well as support for pitch-bend.

5 Conclusion

We presented the `vsti-poly.cpp` FAUST architecture file and its new features: polyphony, pitch-bend and note-history. We used these features in the implementation of a polyphonic sawtooth synthesizer with pitch-bend and portamento slide support, and demonstrated it in a short musical loop, recorded in a popular DAW. We also suggest ideas for further development of VSTi support in FAUST which will contribute to easier implementation of common synthesizer features. The ideas presented here are not limited to the VSTi architecture but could also serve as a reference for implementing FAUST architectures for other plugin formats.

References

MIDI Manufacturers Association. MIDI messages. `http://www.midi.org/techspecs/-midimessages.php`.

Albert Gräf. Interfacing Pure Data with FAUST. In *Proc. 5th Int. Linux Audio Conf. (LAC-07), TU Berlin, http://www.kgw.tu-berlin.de/~lac2007/-proceedings.shtml*, 2007. `http://www.grame.fr/ressources/-publications/lac07.pdf`.

Albert Gräf. Creating LV2 plugins with Faust. In *Proc. 11th Int. Linux Audio Conf. (LAC-13), Graz, http://lac.linuxaudio.org/*, 2013. `http://wiki.faust-lv2.googlecode.com/-hg/faust-lv2-lac13-full.pdf`.

MuTools. Mulab, `http://www.mutools.com/`. `http://www.mutools.com/-mulab-product.html`.

Yann Orlarey, Dominique Fober, and Stephane Letz. Faust: an efficient functional approach to DSP programming. *New Computational Paradigms for Computer Music*, 2009.

Yann Orlarey, Albert Gräf, and Stefan Kersten. DSP programming with FAUST, Q and SuperCollider. In *Proc. 4th Int. Linux Audio Conf. (LAC-06), ZKM Karlsruhe, http://lac.zkm.de/2006/proceedings.shtml*, pages 39–40, 2006. `http://lac.zkm.de/-2006/proceedings.shtml#orlarey_et_al`.

Renoise. Renoise, `http://www.renoise.com`.

Wikipedia. Virtual studio technology.

Latency Performance for Real-Time Audio on BeagleBone Black

James William TOPLISS **Victor ZAPPI** **Andrew McPHERSON**
Centre for Digital Music, School of EECS
Queen Mary University of London
Mile End Road
E1 4NS London,
United Kingdom,
j.w.topliss@se10.qmul.ac.uk victor.zappi@qmul.ac.uk a.mcpherson@qmul.ac.uk

Abstract

In this paper we present a set of tests aimed at evaluating the responsiveness of a BeagleBone Black board in real-time interactive audio applications. The default Angstrom Linux distribution was tested without modifying the underlying kernel. Latency measurements and audio quality were compared across the combination of different audio interfaces and audio synthesis models. Data analysis shows that the board is generally characterised by a remarkably high responsiveness; most of the tested configurations are affected by less than 7ms of latency and under-run activity proved to be contained using the correct optimisation techniques.

Keywords

Embedded systems, BeagleBone Black, responsiveness, latency, real-time.

1 Introduction

Research in Music Technology and, in particular, on Digital Musical Instruments (DMIs) is strongly connected to the field of Human-Computer Interaction (HCI). Following the trend of many other disciplines involving HCI, like Ubiquitous Computing [Kranz et al., 2009] and Augmented Reality [Langlotz et al., 2012; Ellsworth and Johnson, 2013], DMI research has recently started capitalising on portable and embedded systems rather than on general purpose architectures. After many years of complete synergy, musical instruments are increasingly abandoning the laptop/desktop computer in favour of onboard audio processing, leaving an important mark in both academia [Berdahl et al., 2013; Oh et al., 2010; Bacławski and Jackowski, 2013] and industry (e.g., Aleph[1], ToneCore DSP[2] and OWL[3]).

This is due to the fact that DMIs require a specific set of design features to provide the user (i.e., a performer, a composer, a casual player) with a musical experience not too far from the one typical of acoustic and electric instruments [Berdahl and Ju, 2011]. This natural comparison with well known "devices", such as piano and guitar, underlines qualities like reconfigurability, independence/autonomy and high responsiveness, which can be assured only on a dedicated system.

As designers and developers of open source novel DMIs, we have decided to explore the promising and evolving world of embedded Linux technologies, focusing as starting point on the concept of *responsiveness*. The work here presented shows the result of a series of tests aimed at measuring the latency of a BeagleBone Black[4] board (BBB), used as the core of a self-contained, open-source musical instrument. Different hardware and software configurations based on the same Linux kernel (v3.8.13) have been analysed under different CPU loads and levels of code optimisation.

This work is part of a larger structured study, whose goal is to assess longevity, usability and reconfigurability of DMIs, compared to the standards of acoustic and electric musical instruments.

2 Related Work

In 2011 Berdahl et al. presented the *Satellite CCRMA* [Berdahl and Ju, 2011], a platform for teaching and practicing interaction design for diverse musical applications, completely based on embedded Linux. It runs on a BeagleBoard[5] coupled with an Arduino Nano[6] and a breadboard, to support the use of sensors and actuators. Two years later, Berdahl et al. upgraded the platform enabling the compatibility with more powerful boards, such as Rasp-

[1] http://monome.org/aleph/
[2] http://line6.com/tcddk/
[3] http://hoxtonowl.com/
[4] http://beagleboard.org/products/beaglebone%20black
[5] http://beagleboard.org/Products/BeagleBoard
[6] http://arduino.cc/en/Main/arduinoBoardNano

berry Pi[7] and BeagleBoard-xM[8], and expanded the range of possible applications including networking capabilities and hardware-accelerated graphics [s[Berdahl et al., 2013]. The project is based on a Fedora distribution with a custom low latency kernel.

A good example of how new generation embedded Linux boards can be used to extend the capabilities of a musical instrument has been recently presented by MacConnell et al. [MacConnell et al., 2013]. This work introduces a BBB-based open framework for autonomous music computing eschewing the use of the laptop on stage. Some important features of embedded systems are here used to provide the instruments designed using the framework with high degrees of autonomy and reconfigurability. Authors include also data regarding the latency of the system running Ubuntu 12.04. Since mainly FX-like processes are addressed, only the audio-throughput time is measured, under different system load configurations. Results vary between 10 to 15 ms, according to the kind of filtering.

The necessity of measuring the responsiveness of computer-based systems is not recent at all, especially in the context of real-time operative systems. In 2002, Abeni et al. used a series of micro-benchmarks to identify major sources of latency in the Linux kernel [Abeni et al., 2002]. They also evaluated its effects on a time-sensitive application, in particular an audio/video player. Moving towards computer-based audio systems, it is worth mentioning the work by Wright et al. [Wright et al., 2004], in which the latency of MacOS, Red Hat Linux (with real-time kernel patches) and Windows XP are compared, both in an audio-throughput configuration and in an event audio-based configuration. The technique used to estimate the event audio latency consists of measuring the time delay between the sound produced by pressing a button on the keyboard and a sinusoidal audio output triggered on the computer by pressing the button itself.

3 System Configuration

The tests presented throughout this work aim at the evaluation of the capabilities of a BBB-based system when used as development platform for DMIs. The chosen configuration includes most of the standard components required to synthesise and control audio in real-time on a self-contained instrument (i.e., without the need of laptops and any additional external devices). Details on each of these components are given in the following subsections.

3.1 Board and OS

The BBB is an embedded Linux board based on a 1GHz ARM Cortex-A8 processor. It is shipped with an embedded Angstrom Linux distribution (v3.8), optimised to run on embedded architectures. This distro is meant to run general purpose applications and it is not specifically audio-oriented. Our first intent was to explore the capabilities of this default board configuration, without introducing any changes in the underlying kernel. We believe this approach could be very useful for the community of embedded developers to have a clear outline of the built-in audio capabilities of the BBB, thus helping choose the right board.

Although belonging to a new generation of compact and fully accessorised boards (e.g., HDMI, uSD card slot), the BBB does not natively provide any audio interfacing. To test the performances of this board in relation to high quality audio synthesis, two commercially available audio interfaces were chosen for comparison; these were both configured for use with the board so as to provide real-time audio output.

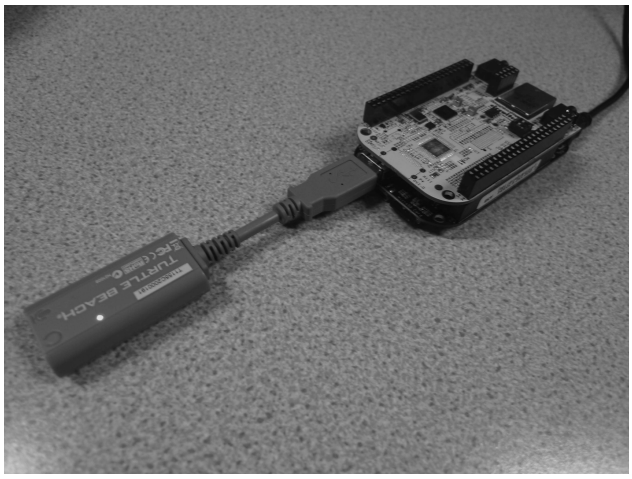


Figure 1: The USB interface and the Audio Cape attached to the BeagleBone Black.

3.2 Audio Interfaces

We tested one USB interface and one BeagleBone expansion "cape" providing audio output.

[7] http://www.raspberrypi.org/

[8] http://beagleboard.org/Products/BeagleBoard-xM

This choice aimed at comparing the two most common solutions used by BBB users for generating audio output. Figure 1 shows both the interfaces attached to the board.

The first interface used is the Turtle Beach Amigo II USB Interface[9]'This device is bus-powered and USB 2.0 class-compliant.. Once attached, this device is automatically recognised as a new hardware interface, meaning that it simply requires being specified as the selected device for audio applications. This interface provides 2 stereo 3.5mm jack receptors, one for input the other for output. Only the latter has been used for our tests.

The second interface used for our tests is the BeagleBone Audio Cape[10]; this device effectively acts as an extension to the BBB, it simply attaches to the top of the board to provide an audio interface. Audio data are exchanged to and from the BBB using an I2S connection. Unlike the USB interface, this device requires some manual configuration to be recognised as a plug-in hardware interface. The on-board HDMI audio virtual cape must be disabled so that the Audio Cape can be loaded by the firmware as the main audio device; this can be easily done by changing the uBoot parameters passed at boot-time. As the USB interface, this cape includes a couple of stereo 3.5mm input/output jack receptors.

3.3 Audio Synthesis

Two different audio backend systems were developed in C++ and cross-compiled to run on the ARM Cortex-A8 processor, one based on ALSA, the other based on JACK. ALSA and JACK implementations are currently adopted by a large number of Linux audio developers.

The audio backend system based on ALSA (Advanced Linux Sound Architecture[11]) essentially comprises an audio engine and a parametric synthesizer. The synthesizer produces frame data; it is connected to the audio engine, which is responsible for collecting and transporting this frame data to the selected output device.

The audio backend system based on JACK (Jack Audio Connection Kit[12]) was similarly designed. A fundamental difference between these two APIs is that JACK uses a client-server model between operating processes and output devices. For this reason, only a synthesizer class was designed to operate as the client process, while a standard JACK server acts as the audio engine for transport. In this configuration the server pulls audio from the client process every time it requires new output data, this is in stark contrast to the ALSA system whereby audio is pushed to the output devices.

Concerning audio synthesis, both the synthesizers implemented in ALSA and JACK generate simple sine waves based on reading from a wavetable. Both systems are configured to provide CD quality audio, (i.e., 16bit resolution, 44.1KHz sample rate) and to run the audio thread at the maximum priority level using a real-time FIFO scheduling.

A parallel control thread was included to manage user input through the keyboard and to have access to the general-purpose in/out pins (GPIO) read/write capabilities of the BBB.

4 Performance Test

The responsiveness and the audio quality of four different specific configurations were tested, combining the use of the 2 audio backends (ALSA an JACK) with the 2 audio devices (USB and Audio Cape). Responsiveness was evaluated considering the latency occurring between the triggering of an audio task producing a waveform and the actual output of the waveform through the audio interface; the assessment of audio quality was connected to the incidence of under-runs.

The performances of each configuration were measured running the audio task in 3 distinct test scenarios. Each of these scenarios (described in the following subsection) involved testing different period and buffer size configurations. As the focus of the test is concerned with very low latency, only the smallest possible period and buffer sizes were examined. In addition, each measurement was repeated enabling 3 different optimisation settings on the C++ cross-compiler (Linaro GCC 4.7 hosted on a x86_64 architecture), using the O1, O2 and O3 flags.

4.1 Test Scenarios

The first scenario involved the generation of a simple monophonic tone. As mentioned in Section 3.2, a simple lookup table was used to generate the frames for this tone.

The second scenario consisted of creating the same monophonic tone as used in the first sce-

[9] www.turtlebeach.com
[10] http://elinux.org/CircuitCo:Audio_Cape_RevA
[11] www.alsa.opensrc.org
[12] www.jackaudio.org

nario, however whilst a `Top` background process was active with a fast refresh rate (passing the command line argument "-d 0.1") (but with standard priority). This scenario allowed for the efficiency of audio synthesis to be observed and measured whilst the system was under heavier load.

The final test scenario was concerned with the generation of a more complex polyphonic tone; this was achieved through the summation of three harmonically related monophonic oscillators. The addition of these two extra tones was considered to be a suitably harder task to synthesise than the simple monophonic tone. It must be noted that no background process was executed during this scenario.

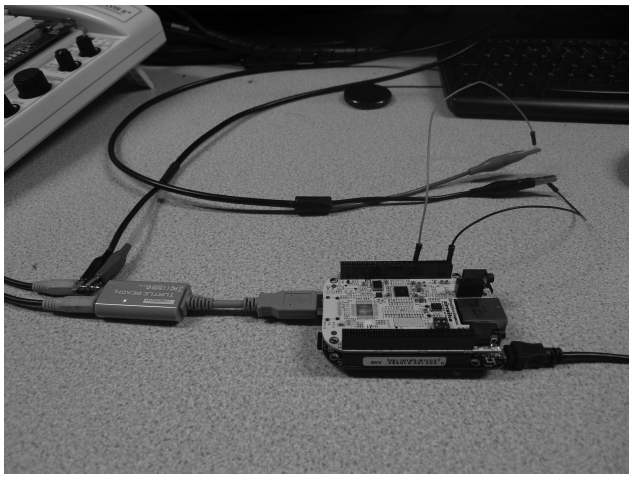


Figure 2: The setup to measure latency when using the USB interface.

4.2 Procedure

The BBB was connected by USB; tests were performed over an ssh connection via BBB's USB network connection. One of the board's GPIOs was attached to the first input channel of an oscilloscope. The audio output was connected to the second channel of the oscilloscope. For the case of the USB interface, the complete setup is shown in Figure 2.

In detail, the test procedure ran as follows. Upon starting one of the executables, the generated system (ALSA or JACK) was programmed to initialise itself but then wait for user input (i.e. a keystroke) before beginning to fill the output buffers with frames. Once the keystroke signal was received across the serial connection, the first task of the system was to drive the GPIO connected to the oscilloscope from low to high. Only immediately after this the audio cycle could begin, outputting the signal into the oscilloscope. The oscilloscope was set to trigger a single display capture on both the channels upon the detection of a rising edge in the GPIO signal. The time distance between the GPIO rising edge in the display and the beginning of the captured audio output hence provided a measurement of the operational latency (Figure 3); each measurement was repeated 5 times [assuming that's right] and an average value calculated.

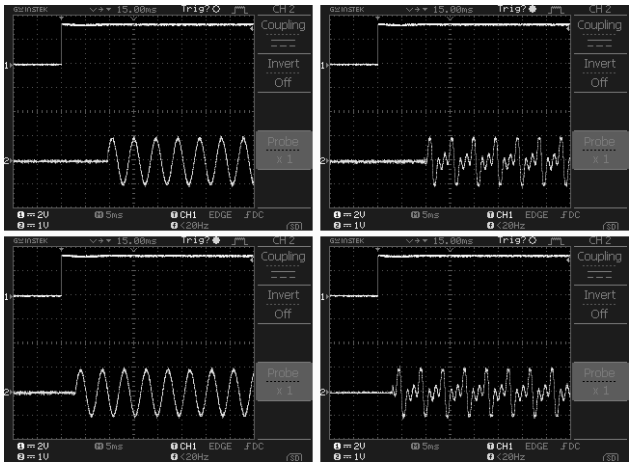


Figure 3: Examples of oscilloscope display capture for scenarios 1 and 3. Latency is measured as the horizontal distance (time gap) between the GPIO signal rising edge (in yellow) and the start of the waveform (in light blue).

It must be noted that the period and buffer sizes chosen were dependent upon both the system type and the target interface; configurations that worked well for one pair did not necessarily run well for another. The six smallest usable configurations were tested for each system and interface pair.

In addition to measuring the output latency, the quality of the output audio was also observed; this observation relied upon noting the frequency of frame dropout (under-run activity) and visible distortion displayed on the oscilloscope (if any).

5 Results

The reported measurements are here presented and discussed, first globally and then analysing more specific cases. Both latency and quality of the output (under-runs) are taken into account and the singular contributions are combined.

5.1 Latency

The latency results were highly consistent across trials: when using the USB audio interface with both ALSA and JACK systems (Fgures 4 and 6) the maximum difference between individual measurements generally only varied by one millisecond, with only few exceptions; this was true regardless of the used optimisation. Measurements concerning the Audio Cape (Figures 5 and 7) were even more consistent than for the USB interface. Across the five measured latencies, measurements only varied by half of a millisecond, without exceptions.

As expected, for all configurations, latency is directly related to buffer and period sizes. No significant, systematic latency differences were noted amongst the three optimisation settings.

However, the choice of monophonic versus polyphonic synthesis and the system load introduce unexpected variations in the latency. These variations may reflect a delay in starting up the ALSA or JACK system, rather than a difference in steady-state latency once the audio rendering is running. Our test procedure toggles a GPIO pin and then immediately begins filling the audio buffers; it is possible that this initial startup produces an additional transient delay compared to reacting to an event once audio is already running. In any case, the difference between test conditions is always less than 1ms.

ALSA - USB Interface

Buffer	Period	O1 Optimisation			O2 Optimisation			O3 Optimisation		
		Monophonic	Mon+bgdproc.	Polyphonic	Monophonic	Mon+bgdproc.	Polyphonic	Monophonic	Mon+bgdproc.	Polyphonic
		Average Latency (mS)								
128	64	5.6	5.7	5.9	5.2	5.6	5.4	5.6	5.3	5.5
256	64	6.4	6.7	6.4	6.6	6.5	6.4	6.2	6.2	6.6
512	64	7.8	7.9	8.4	7.9	7.9	8.2	8.0	7.8	7.9
256	128	6.3	6.5	6.7	6.5	6.5	6.5	6.5	6.5	6.6
512	128	8.0	8.2	8.4	8.2	8.4	8.1	8.3	7.7	7.8
512	256	9.8	9.7	10.7	8.1	9.9	11.2	9.9	10.7	9.7

Figure 4: ALSA latency measurements for the USB interface.

ALSA - Audio Cape

Buffer	Period	O1 Optimisation			O2 Optimisation			O3 Optimisation		
		Monophonic	Mon+bgdproc.	Polyphonic	Monophonic	Mon+bgdproc.	Polyphonic	Monophonic	Mon+bgdproc.	Polyphonic
		Average Latency (mS)								
128	8	2.5	2.5	2.5	2.4	2.5	2.5	2.4	2.4	2.5
256	8	3.5	3.5	3.9	3.3	3.5	3.6	3.4	3.4	3.7
512	8	5.6	5.7	6.0	5.7	5.7	5.9	5.5	5.4	5.9
256	16	2.9	2.8	3.0	2.7	2.9	3.0	2.6	2.8	2.9
512	16	4.3	4.5	5.0	4.4	4.2	4.8	4.3	4.5	4.6
512	32	3.5	3.5	4.0	3.5	3.6	4.0	3.5	3.6	4.1

Figure 5: ALSA latency measurements for the Audio Cape.

It can be noted that, on both systems, the Audio Cape allows for smaller buffer and period configurations. Unfortunately, the set of available configurations varies between the 2 systems, making impossible a direct performance comparison. For the only overlapping configuration (buffer 512 - period 32), JACK performed unexpectedly badly, showing a higher latency than configurations based on larger period sizes.

Conversely, USB interface results extend on the same set of configurations for both ALSA and JACK, so that a quantitative comparison is here possible. For the smallest period size (i.e. 64 frames) the JACK system shows better or equal performances, while increasing the size ALSA proved remarkably more responsive. In particular, the last 3 cases listed in Figure 6 shows that JACK's latency is almost the same and quite high, regardless of all the conditions, i.e. buffer/period sizes, test scenario and optimisation level.

JACK - USB Interface

Buffer	Period	O1 Optimisation			O2 Optimisation			O3 Optimisation		
		Monophonic	Mon+bgdproc.	Polyphonic	Monophonic	Mon+bgdproc.	Polyphonic	Monophonic	Mon+bgdproc.	Polyphonic
		Average Latency (mS)								
128	64	4.5	4.8	4.5	5.0	4.3	4.5	4.5	4.7	4.5
256	64	6.3	7.2	6.7	6.3	7.2	6.7	7.3	6.5	6.5
512	64	6.0	6.2	6.5	6.2	6.5	6.5	6.7	7.0	6.2
256	128	11.8	12.5	12.3	11.8	12.7	10.7	12.7	11.7	11.7
512	128	11.3	11.7	12.0	11.2	11.2	11.0	11.8	11.8	11.3
512	256	11.3	11.3	11.2	11.3	11.3	10.8	11.7	11.8	11.2

Figure 6: JACK latency measurements for the USB interface.

JACK - Audio Cape

Buffer	Period	O1 Optimisation			O2 Optimisation			O3 Optimisation		
		Monophonic	Mon+bgdproc.	Polyphonic	Monophonic	Mon+bgdproc.	Polyphonic	Monophonic	Mon+bgdproc.	Polyphonic
		Average Latency (mS)								
128	16	3.0	2.8	2.8	3.0	3.0	2.7	2.7	3.0	2.7
128	32	2.2	2.2	2.3	2.2	2.2	2.0	2.5	2.2	2.2
256	32	5.0	5.0	5.2	5.0	5.0	5.0	5.0	5.0	5.0
512	32	11.2	11.0	11.0	11.0	11.0	11.0	11.0	10.8	11.0
256	64	3.8	4.2	4.2	4.3	4.7	4.3	4.0	4.5	4.0
512	64	9.7	9.8	10.0	10.0	10.0	10.0	9.5	10.2	10.7

Figure 7: JACK latency measurements for the Audio Cape.

5.2 Under-runs

The test highlighted a certain incidence of under-runs, whose effects varied according to the chosen configuration. Generally, they occurred in particular when lowest buffer and period sizes were tested. Also the different optimisation settings proved to strongly affect their incidence.

5.2.1 ALSA

Using ALSA on the USB interface, it was observed the occurrence of frame dropout issue only when the buffer and period were set to the minimum values (i.e. respectively 128 and 8 frames). This was true across all the build qual-

ities of the system and for all of the scenarios. During the first scenario (pure tone) and third scenario (polyphonic tone) observed dropouts were not too severe, normally only being exhibited once or twice at the beginning of synthesis. The second scenario seemed to generate significantly higher rates of frame dropout, leading to audible clicks. An interesting observation is that the O2 optimised versions of the system appeared to always exhibit the least amount of under-runs.

In the case of the ALSA system using the Audio Cape, it was observed that frame dropout only occurred whilst using the smallest period size configurations (period size of 8), regardless of the buffer setting. Again this was true across all the build qualities of the system and all of the scenarios. The first and third scenario produced a very small amount of under-run activity, interestingly only for the first period and buffer size configuration tested. Similarly to the USB interface, these observed dropouts were very minor, normally only being exhibited once or twice at the beginning of synthesis. In the second scenario the amount of frame dropout did increase slightly. Interestingly, this time the O3 optimised versions exhibited the best improvements, almost completely preventing all under-runs.

5.2.2 JACK

Since in JACK the stream to the audio device is not managed by the client, under-runs can occur only not he server. In relation to the JACK system using the USB audio interface, most frame dropout issues observed occurred during the first size configurations (buffer 128 - period 16), the second one (buffer 128 - period 32) and the fourth one (buffer 256 - period 32). In regards to the first and third scenarios, the frame dropouts noted for the first period and buffer size configuration were very severe for the O1 and O2 optimisations. The amount of under-run activity for this configuration made it very difficult to gather any measurements for latency; sometimes the JACK server would under-run continuously without the client even being active. The O3 optimisation however did not experience this problem for this configuration; under-runs were noted however were nowhere near as severe. In the case of the second scenario, far less dropouts were observed consistently across all build qualities, a surprising result. Again, O3 optimisation proved to provide the best performance enhancement.

In the case of the JACK system using the Audio Cape, it was noticed that the occurrence of frame dropout appeared more frequently for the first three period and buffer size configurations. The first scenario produced a very small amount of under-run activity, in regards to all the three optimisations; only during the first scenario were any frame dropouts observed. The nature of these under-runs however was different to those previously observed; during the synthesis the server ran smoothly, while under-runs were noted only after the client had been disconnected. It was observed during the second scenario that the amount of frame dropout increased slightly; the type of under-run seen in the first scenario (after the termination of the client) occurred more frequently. This behavior was exhibited in both the first and second period and buffer size configurations for the O1 and O3 optimisations. In the case of the O2 optimisation, this behavior was not observed; instead a severe under-run issue occurred during the second period and buffer size configurations whereby the server immediately began to under-run before the client had even been launched. In relation to the third scenario, similar types of under-run behavior were seen as in the previous two scenarios whereby under-runs occurred after the termination of the client. No frame dropouts were observed for the O2 optimisation for this particular scenario.

6 Conclusion

Throughout this paper we presented a study aimed at evaluating the responsiveness of a Linux embedded system. As part of a larger study on DMIs design, we focused on testing latency and quality of audio output on a BeagleBone Black board running the standard Angstrom distribution with no kernel modifications. Two different audio backend systems were taken into consideration, one based on ALSA, the other on JACK, and measurements using a USB audio interface and an Audio Cape were compared. The test monitored event-to-audio latency and included the monitoring of under-run activity.

Data analysis showed for both ALSA and JACK audio systems remarkably low latency values, especially for small buffer and period size configurations. In particular, the use of the Audio Cape allows for latency values lower than 3ms. In some of the different CPU scenarios taken into consideration the audio stream pre-

sented dropouts and clicks, especially when using small buffer and period size configurations. However, the usage of the different levels of code optimisation available in the chosen compiler (cross-Gcc O1, O2 and O3) completely fixed the audio quality in most of the tested configurations.

No previous works delved into the audio capabilities of the BeagleBone Black while running the default Linux distribution (Angstrom with kernel 3.8). Compared to other distributions, like Ubuntu or Fedora, the usage of Angstrom on the BeagleBone Black proved to support very low latency configurations without the need of a customised kernel. In the context of digital musical instrument design, this feature is remarkable and makes the BeagleBone Black an appealing platform for instrument development.

References

Luca Abeni, Ashvin Goel, Charles Krasic, Jim Snow, and Jonathan Walpole. 2002. A measurement-based analysis of the real-time performance of linux. In *IEEE Real Time Technology and Applications Symposium*, pages 133–142. IEEE Computer Society.

Krystian Bacławski and Dariusz Jackowski. 2013. Komeda: Framework for interactive algorithmic music on embedded systems. In *Sound and Music Computing Conference*.

Edgar Berdahl and Wendy Ju. 2011. Satellite ccrma: A musical interaction and sound synthesis platform. In *International Conference on New Interfaces for Musical Expression*.

Edgar Berdahl, Spenser Salazar, and Myles Borins. 2013. Embedded networking and hardware-accelerated graphics with satellite ccrma. In *International Conference on New Interfaces for Musical Expression*.

Jeri Ellsworth and Rick Johnson. 2013. Castar - technical illusions. http://technicalillusions.com/?page_id=197.

Matthias Kranz, Albrecht Schmidt, and Paul Holleis. 2009. Embedded interaction - interacting with the internet of things. *IEEE Internet Computing*, 99(1).

Tobias Langlotz, Stefan Mooslechner, Stefanie Zollmann, Claus Degendorfer, Gerhard Reitmayr, and Dieter Schmalstieg. 2012. Sketching up the world: in situ authoring for mobile augmented reality. *Personal and Ubiquitous Computing*, 16(6):623–630.

Duncan MacConnell, Shawn Trail, George Tzanetakis, Peter Driessen, and Wyatt Page. 2013. Reconfigurable autonomous novel guitar effects (range). In *Sound and Music Computing Conference*.

Jieun Oh, Jorge Herrera, Nicholas Bryan, and Ge Wang. 2010. Evolving the mobile phone orchestra. In *International Conference on New Interfaces for Musical Expression*, Sydney, Australia.

Matthew Wright, Ryan J. Cassidy, and Michael F. Zbyszynski. 2004. Audio and gesture latency measurements on linux and osx. In *International Computer Music Conference*, pages 423–429, Miami, FL. International Computer Music Association.

Mephisto: an Open Source WIFI OSC Controller for Faust Applications

Romain MICHON
Center for Computer Research in Music and Acoustics
Department of Music,
Stanford, CA 94305-8180,
USA,
rmichon@ccrma.stanford.edu

Abstract

MEPHISTO is a small battery powered open source Arduino based device. Up to five sensors can be connected to it using simple 1/8" stereo audio jacks. The output of each sensor is digitized and converted to OSC messages that can be streamed on a WIFI network to control the parameters of any FAUST generated app.

Keywords

Faust, Arduino, Controller, OSC

1 Introduction

In the past few years, the FAUST[1] [Orlarey and Letz, 2002] programming language has been used increasingly by researchers and developers to implement new algorithms for real time audio signal processing. As a result, dozens of open source FAUST effects and synthesizers are now freely available. For example, Julius Smith's libraries [Smith, 2012] and the FAUST-STK [Michon and Smith, 2011] provide a large array of objects ranging from the simplest lowpass filter to complex feedback delay networks and physical models of musical instruments.

However, we observed that these technologies remain relatively inaccessible to musicians who don't have the knowledge (and the desire) to compile a FAUST object on their laptop. In other words, these elements are not "plug and play".

One of the tool already at our disposal to facilitate the sharing and the use of FAUST objects is the Online Compiler[2] [Michon and Orlarey, 2012]. This web app contains an interactive catalog of FAUST programs that can be compiled to any of the available FAUST architectures and then downloaded. Users can easily add their own FAUST codes to the catalog or modify existing elements. Even if this very high level tool makes the creation of plug-ins, etc., very easy, it targets users who have some knowledge in computer music and who know how to use a VST[3], an external audio interface, etc.

Thus, to make things even easier we started to think about a FAUST stomp box that could be based on an embedded Linux system such as a *Raspberry Pi*[4]. It would have been able to connect to the online compiler to provide its user a list of the objects stored in the catalog. A download button to cross compile and then download the effect or the synthesizer in the FAUST box would have made the use of this system very easy.

However, even though *Raspberry Pis* are great prototyping platforms, their computation power is quite limited. Also, their booting time can be a problem for impatient musicians. We then realized that a smartphone or a tablet could do a similar job and would be more user friendly. While there already existed a `faust2ios` architecture, Apples product were presenting a huge disadvantage over Android phones: in order to be installed, an app has to be approved by the *Apple Store* which was making our concept of a customizable stomp box impossible to implement. Thus we opted for Android and created a `faust2android` [Michon, 2013] architecture.

Another key component of our project was to provide an easy way for musicians to control the different parameters of a FAUST object during a live performance. While smartphones offer built-in basic controllers: touch screen, accelerometer, etc., these elements are not very practical to interact with if the user is playing an instrument and processing its sound with his phone. Indeed, many instruments require the use of both hands, making it inconvenient to interact with another interface.

MEPHISTO was created to solve this problem.

[1] `http://faust.grame.fr`.
[2] `http://faust.grame.fr/compiler`.
[3] *Virtual Studio Technology*.
[4] `http://www.raspberrypi.org/`.

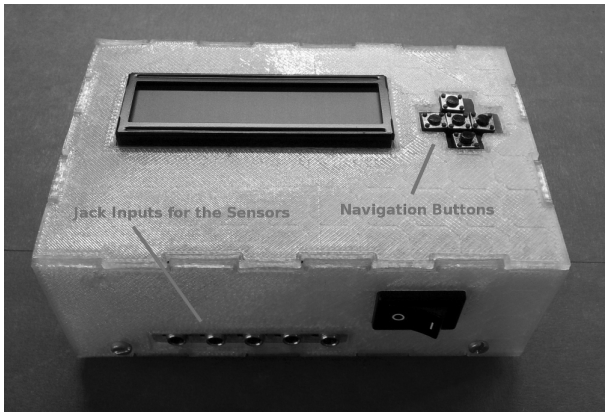


Figure 1: View of MEPHISTO from its top.

It is a small battery powered device that can be easily attached on someone's belt (cf., figures 1 and 2). Up to five sensors can be connected to it using simple 1/8" stereo audio jack plugs. The output of each sensor is digitized and converted to OSC[5] [Wright, 2005] messages that can be streamed on a WIFI network to control the parameters of any FAUST generated app. As OSC is a standard protocol, MEPHISTO can be used with `faust2android` apps but is also compatible with most of the FAUST architectures and programs enabling OSC communication.

As a "DIY"[6] open source project, MEPHISTO only uses open source hardware (Arduino, etc.) and was designed to be easily built by anyone. A web page giving the instructions to build your own MEPHISTO has been created[7].

2 Hardware

Designing small scale open source hardware can be a rather complicated task. Indeed, while software can be easily deployed and shared, in many cases hardware requires a production chain, etc. For this reason, MEPHISTO has been designed to be easily and quickly built by anyone.

2.1 The Case

To make it as easy as possible for users, the case of MEPHISTO is 3D printable. It has been designed with Blender[8] which is an open source program for 3D design that has some CAD features.

[5] *Open Sound Control*: `http://opensoundcontrol.org/`.

[6] Do It Yourself.

[7] `http://ccrma.stanford.edu/~rmichon/mephisto`.

[8] `http://www.blender.org/`.

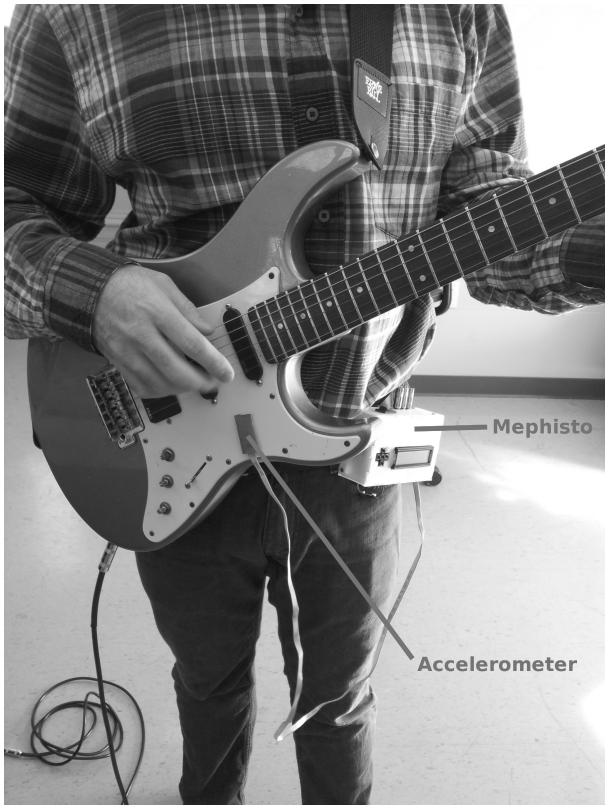


Figure 2: Use example of MEPHISTO.

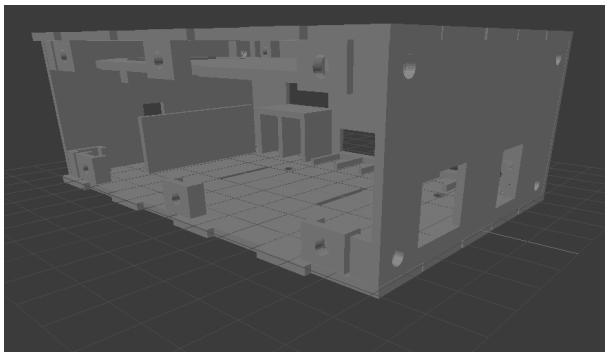


Figure 3: 3D model of MEPHISTO's case as it appears in `Blender`.

We're perfectly aware that not everyone has a 3D printer at home, but 3D printed models can now be ordered very easily online for very cheap. Moreover, MEPHISTO's case has a very simple design and can be printed on almost every 3D printers.

2.2 Electronics

MEPHISTO is based on an Arduino Uno[9] and a WIFI Shield[10] (cf., figure 4). The Arduino provides five analog inputs that are used in

[9] `http://arduino.cc/en/Main/arduinoBoardUno`.

[10] `http://arduino.cc/en/Main/ArduinoWiFiShield`.

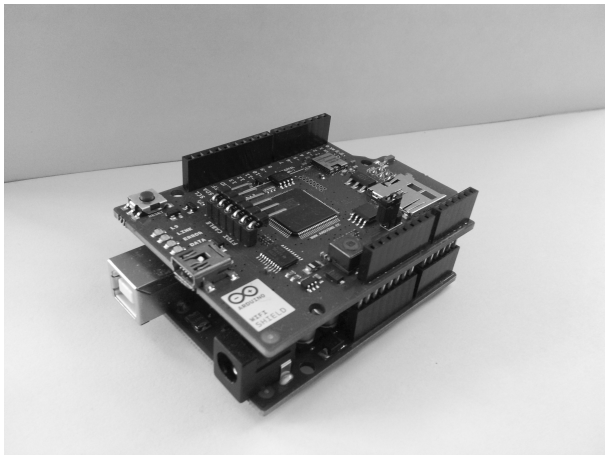


Figure 4: An Arduino Uno and its WIFI shield.

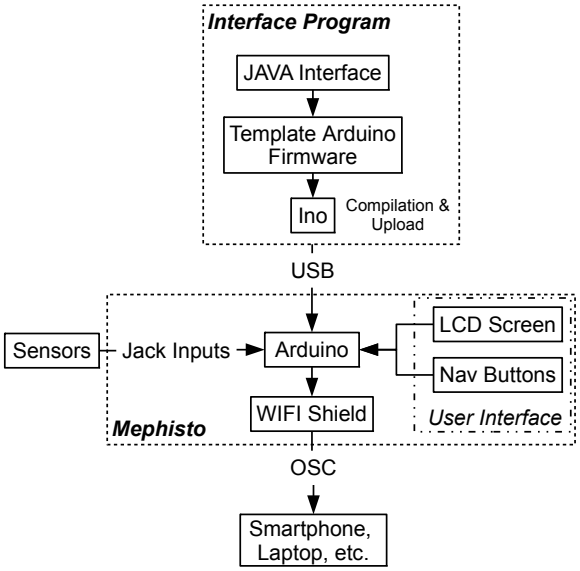


Figure 5: Mephisto flow chart.

Mephisto to digitize the output signals of the sensors. Simple 1/8" stereo audio jacks (three pins) are used to bring power to the sensors and to retrieve their output signal (cf., figure 1).

Users can configure basic parameters such as the WIFI network to connect to or the IP address of the host using an LCD screen and a navigation button interface.

Mephisto can be powered with any DC power adapter between seven and nine volts or with a simple nine volts battery. We considered using lithium ion batteries instead but these are more expensive and need a special charger. With five simple sensors plugged to it, Mephisto can run for about four hours on the same nine volts battery. Moreover, it is very easy and quick to replace it.

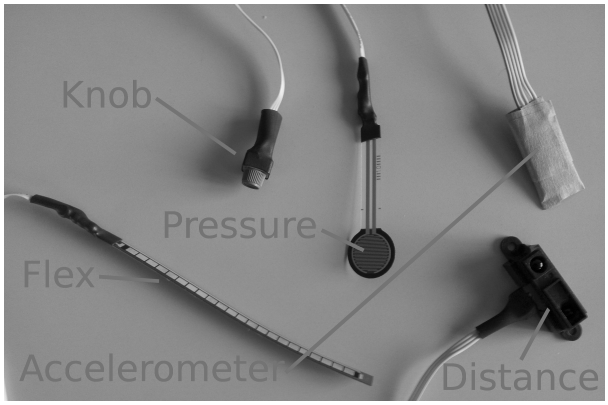


Figure 6: Example of sensors that can be connected to Mephisto with their 1/8" jack plugs.

Dozens of sensors have been tested and can be easily prepared to work with Mephisto. Our website explains how to set up an accelerometer, a pressure sensitive and a flex sensors, trim pots, etc.[11].

3 Software

3.1 Arduino Firmware

The Arduino firmware carries out a large number of tasks. It retrieves and digitizes the output signals of the sensors and scale them in function of the parameters specified in the interface program (cf., §3.2). Then it converts them to OSC messages using `oscuino`[12]. The OSC address of each sensor can be configured in the interface program (cf., §3.2).

The firmware also handles the user interface implemented through the LCD screen and the navigation buttons.

3.2 Interface JAVA Program

Even though Mephisto provides its own very simple interface to configure it by the mean of its LCD screen and navigation buttons, a `JAVA` program[13] that can run on both Linux and MacOSX was created to carry out this task more precisely.

This simple program allows to configure the OSC address and the range of the OSC messages sent by Mephisto for each jack input. It is also possible to choose which sensor is connected to which jack in order to carry out some scaling on their output signal (even if the Mephisto website explains how to do basic electronic scaling

[11] `http://ccrma.stanford.edu/~rmichon/mephisto`.
[12] `http://cnmat.berkeley.edu/oscuino`.
[13] `https://github.com/rmichon/mephisto/`.

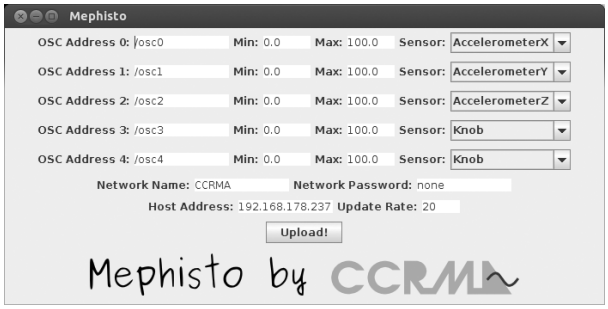


Figure 7: Screenshot of the interface program used to configure MEPHISTO from a desktop.

on sensors, it is often necessary to adjust their output range computationally).

The MEPHISTO configuration program also makes it possible to pre-configure the WIFI network to which mephisto will connect as well as its password if it is protected, the IP address of the host and the rate at which the messages are sent.

This interface program formats and customizes the Arduino firmware in function of the provided parameters. It then compiles it and uploads it to the Arduino if it is connected to one of the USB port using `ino`[14]. As this program only works with Linux and MaxOSX it makes the interface only usable on these platforms even though it can also be executed on Windows.

4 Conclusion

MEPHISTO is an open source project that improves and simplifies the control of sound effects and synthesizers running on a mobile device. Any kind of sensor can be connected to it and used as on OSC controller for live performance.

The FAUST online compiler together with MEPHISTO, `faust2android` and the FAUST catalog of sound effects and synthesizers greatly simplifies the use of FAUST objects by musicians.

The use of 3D printing in the framework of open source hardware projects makes things a lot easier than in the past. Indeed users don't need to have any background in manufacturing and only have to take care of putting the different components together.

Similarly, Arduinos are relatively self contained environments that significantly reduce the size of electronic circuits making projects like MEPHISTO easy to build at home.

[14] `http://inotool.org/`.

References

Michon and Orlarey. 2012. The faust online compiler: a web-based ide for the faust programming language. In *Proceedings of the Linux Audio Conference (LAC-2012)*, pages 111–116, Stanford University, USA.

Michon and Smith. 2011. Faust-stk: a set of linear and nonlinear physical models for the faust programming language. In *Proceedings of the Conference on Digital Audio Effects (DAFx-11)*, pages 199–204, IRCAM, Paris, France.

R. Michon. 2013. Faust2android: a faust architecture for android. In *Proceedings of 16th Int. Conference on Digital Audio Effects (DAFx-13)*, pages 301–304, National University of Ireland, Maynooth, Ireland.

Fober Orlarey and Letz. 2002. An algebra for block diagram languages. In *Proceedings of the International Computer Music Conference (ICMA)*, pages 542–547, Gothenburg, Sweden.

J. Smith. 2012. Signal processing libraries for faust. In *Proceedings of the Linux Audio Conference (LAC-2012)*, pages 33–38, Stanford University, USA.

M. Wright. 2005. Open sound control: an enabling technology for musical networking. *Organised Sound*, 10(03):193–200.

Processes in real-time computer music

Miller Puckette
Department of Music
University of California, San Diego
msp@ucsd.edu

Abstract

The historical origin of currently used programming models for doing real-time computer music is examined, with an eye toward a critical re-thinking given today's computing environment, which is much different from what prevailed when some major design decisions were made. In particular, why are we tempted to use a process or thread model? We can provide no simple answer, despite their wide use in real-time software.

Keywords

real time, computer music, processes, parallelism

1 Introduction

The language of real-time computer music borrows from three antecedents that were fairly well in place by 1985, before the field of real-time computer music took its current form. Classical computer music models, starting with Max Mathews's MUSIC program (1957), were well studied by that time. The field of computer science, particularly operating system design, was also taking shape; perhaps it may be said to have matured by 1980 with the widespread adoption of Unix. Meanwhile, a loosely connected network of electronic music studios arose in the 1950s whose design is directly reflected in the patching paradigm that is nearly universal in modern computer music environments.

Both computer science and music practice relied on a notion of parallelism, albeit in very different forms and terms. In computer science, abstractions such as *process* and *thread* arose from the desire to allocate computing resources efficiently to users. In music, thousand-year old terms like *voice* and *instrument* imply parallelism, both on written scores as multi-part music were notated for both practical and dogmatic reasons, and in real time as live performers sang or played the music in ensembles.

In both computer science and computer music language, abstractions modeled on processes or threads are used to try to describe the passage of time and also to express, and/or take advantage of, parallelism. But the aims in computer science (efficiency) are different from those in computer music (as an aid to organizing musical computation).

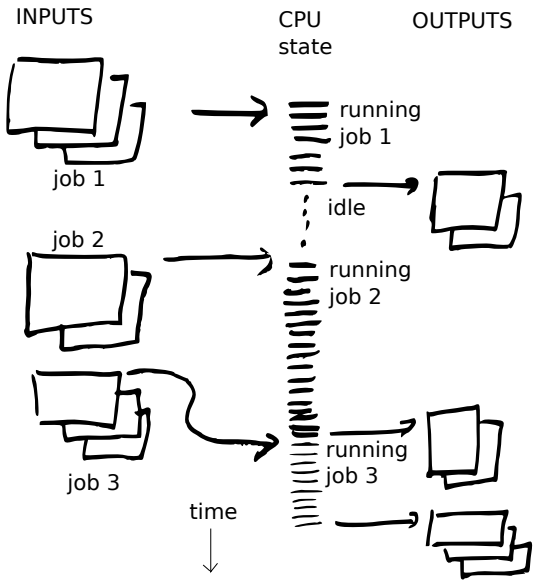


Figure 1: Submitting jobs to a computer circa 1960.

In the sections that follow I will try to trace these developments historically to see why we treat processes and related concepts in the way that we now do in real-time computer music systems. I hope to help clarify why the current practice is what it is, and perhaps contribute to thinking about future computer music programming environments..

2 Computer science terminology

In classical operating system design theory, the tasks set before a computer were organized into *jobs*. A prototypical computer (fig. 1) sat in a room waiting for jobs to be submitted to it, per-

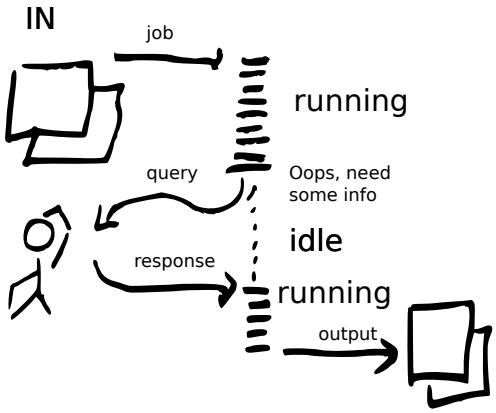


Figure 2: Interactive jobs may stall in mid-computation to ask the operator for more information.

haps in the form of stacks of punched cards. The computer would execute each job in turn, hopefully producing output which could also have been stacks of punched cards. At least three problems arise in this scenario:

- **Idleness.** The computer sometimes had nothing to do and would be idle; idle time reduced the total amount of computation the computer could carry out.
- **Latency.** Sometimes a job would be submitted while another job was running (as in job number 2 in the figure); in this case the job would join a queue of waiting jobs. This meant that the submitter of job 2 had to wait longer to harvest the output.
- **Unanticipated data needed.** For many types of jobs you might not be able to predict at the outset what data will be needed during the computation. The "job" model doesn't offer a way for the computer to ask the operator for additional information it might need.

The first two of these only impact the efficiency of computation, but the third requires that we go back and amend the job model altogether; so we will consider that first. Figure 2 shows an amended model of computation allowing *interactive* jobs that may stop execution part way through and ask the operator for more information. When this happens the job is considered *stalled* and the computer sits idle.

Computer science's answer to the problems of idleness and latency have been to intro-

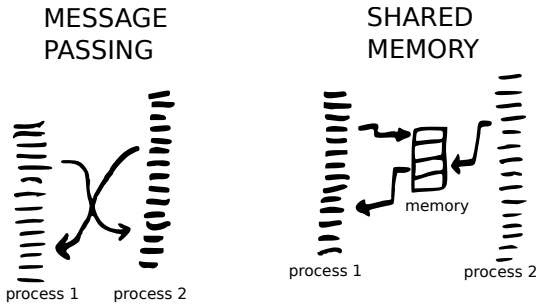


Figure 3: Process intercommunication using messages and shared memory.

duce *time-sharing* and *multiprocessing*. Time-sharing is the practice of keeping several jobs in progress at the same time, so that when one job stalls or finishes, the processor's time can then be devoted to some other job that needs running. Perhaps this second job will later stall or finish but meanwhile, too, the first job may have become runnable again (having received a new dose of data it had stalled waiting for). The computer would then return to job 1. One could also fill idle time by keeping low-priority jobs waiting in the background (ones whose latency requirements were less strict) that would run whenever all higher-priority jobs were stalled.

The advent of multiprocessors made it possible to further improve throughput in the same way that having several short-order cooks in a diner can speed orders. As the number of jobs and the number of available processors increases, there should be fewer wild swings in the availability of processing power to satisfy the needs of submitted jobs.

The chief tool for time-sharing and multiprocessing is an abstraction called a *process*, which can be thought of as a virtual computer. When a job is submitted, one creates a brand new (virtual) computer to carry it out, and once the job is finished, the virtual computer, or process, is scrapped. Each job may run in ignorance of all other jobs on the system. Each process gets its own memory and program to run, and its own *program counter*, abbreviated *PC*, that records where in the program the computer is now running. When the computer switches from running one process to another one, the memory and PC (and other context) of the first process are retained so that they are available again when the first process is again run in the future.

Although at the outset we could consider all processes to operate in complete ignorance of each other, at some point the need will certainly

arise for processes to intercommunicate. Computer science offers at least two paradigms that we will want to consider: *message passing* and *shared memory* (see fig. 3). Of these, the message passing paradigm is less general but easier to analyze and make robust. In message passing, one process can simply send another a packet or a stream of data, that the second one may read at any later time. This is similar conceptually to how people intercommunicate. The chief difficulty using this paradigm is that it does not allow a process to interrogate another directly, except by sending a message and then stalling until a return message is received. This might greatly increase the latency of computations, and worse yet, if we adopted this strategy for interrogation, two processes could conceivably interrogate each other at the same time, so that both end up deadlocked.

In the shared-memory paradigm two processes communicate by reading and writing to a shared area of memory. We can then arrange for one process to be able to interrogate another one simply by looking in the appropriate location in its memory (which, by prior arrangement, we had arranged to share). But now we have to work hard to make sure that our two processes will carry out their computations deterministically, because the order in which the two access the shared memory is not controlled. We would need to set up some convention to manage this. (One such convention could be to format the shared memory into message queues, thus returning us to the situation described in the previous paragraph.) In general, there is no final answer here; any paradigm will either be onerously restrictive or dangerously permissive, or both, and to make good choices will require careful attention to the particulars of the task at hand.

3 Electronic music terminology

The first widely used model for computer music performance was what is now called *Music N*, developed over a series of programs written by Max Mathews starting in 1957[Mathews, 1969]; by 1959 his Music 3 program essentially put the idea in its modern form, as exemplified in Barry Vercoe's Csound program. These programs all act as "music compilers" or "renderers", taking a fixed text input and creating a soundfile as a batch output. Although Csound has provisions for using real-time inputs as part of its "rendering" process, in essence the programming model

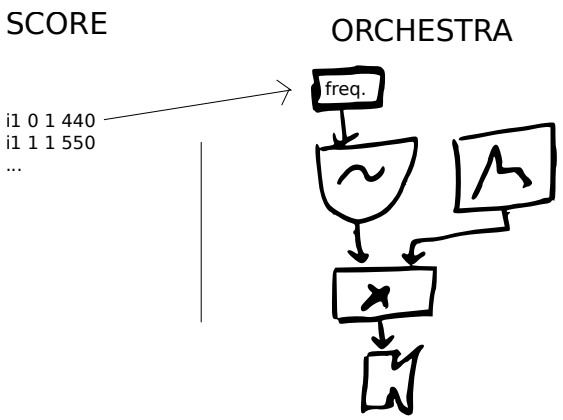


Figure 4: The Music N paradigm

is not interactive.

Music N input is in the form of an *orchestra* and a *score*, as shown in fig. 4. The orchestra can be thought of as emulating a 1950s-era *elektronischemusik* studio, in which the hardware is organized in metal boxes with audio inputs and outputs, such as tape recorders, oscillators, pulse generators, filters, ring modulators, and so on. These would be connected by audio cables into a *patch*. Furthermore, the boxes had knobs and switches on them that allowed the user to supply parameters such as the frequency of an oscillator.

In the Music N paradigm, the studio and its patch are represented by the orchestra. Although the actual orchestra file is in a programming language, when publishing algorithms it has traditionally been represented as a block diagram showing a collection of *unit generators* and the audio connections between them.

The score is organized as a list of time-tagged records that are (either nostalgically or deprecatingly) called *score cards*. In addition to one or two time tags (a "note" has two, one for its start and one for its end), a score card has some number of numerical parameters that may be supplied to the unit generators. The score is like a process in that it runs sequentially in time. Unlike the computer science notion of a process, however, the score advances and waits according to timing information in the score cards. Each score card has an associated *logical time* at which it is run.

Things get interesting when we try to adapt this paradigm to run in real time. We could simply connect the Music N output to a real-time audio output; but presumably our reason

for wanting to run in real time is to be able to use live inputs to affect the sound output. Taking the opposite direction, we could require that the user or musician supply all the parameters in real time using knobs and switches, but this quickly reveals itself to be unmanageable for the human. We will need to make intelligent decisions, probably different for any two musical situations, as to how the live inputs will affect the production of sound. More generally, our problem is to design a software environment that will give a musician the freedom to make these choices.

In the early 1980s two influential real-time synthesizers were designed, the Systems Concepts Digital Synthesizer (or "Samson Box") at Stanford[Loy, 1981], and the 4C synthesizer at IRCAM[Moorer et al., 1979][Abbott, 1981]. Both machines ran a fixed computation loop with a fixed number of steps, with one loop finishing at each tick of the sample clock,

Each of these machine designs got some things right for the first time. The Samson box was the first working machine that could do sample-accurate parameter updates in real time. To do this, the fixed program contained an update mechanism in which items were taken off the head of a time-tagged parameter update queue. This queue was filled by the Foonly controlling computer some tenths of seconds, or whole seconds, in advance, so that the Foonly did not have to preform parameter updates synchronously. This approach had one major limitation: it did not take into account the possibility of real-time interaction. It was physically possible to jump the queue for "real-time" parameter updates, but then one lost any ability to determine the timing of such updates accurately.

The 4C machine and its controlling software 4CED were more explicitly designed with real-time interaction in mind, although the timing was less accurate than with the Samson Box. In the 4C parameter updates were effected at interrupt level from the controlling computer; the computer was interrupted by the 4C when one of a bank of timers ran out.

The 4CED user conceptualized the 4C as a collection of 32 independent processes (Abbott's simile was a collection of 32 music boxes that the user could start at any time). The guiding idea seems to have been that a performer could play a keyboard with 32 keys on it, but each key, rather than being restricted to playing a single

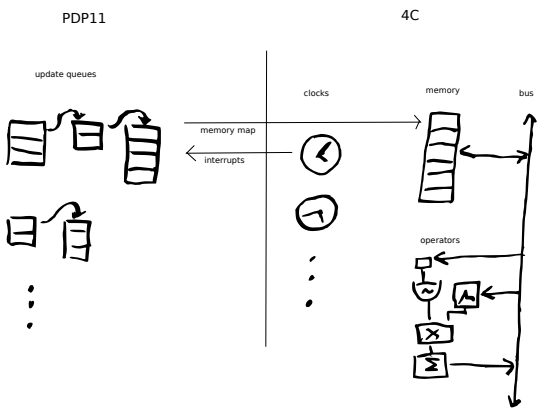


Figure 5: The 4C and the 4CED environment

note, could in turn set off a whole sequence of actions. This would seem to greatly magnify what the keyboard player could do.

It seems also to have been on people's minds that the playing of sequences could usefully be unified with the business of scheduling breakpoints to a pitch or amplitude envelope consisting of many segments. Both the sequencing of collections of notes and the sequencing of envelope breakpoints led many computer music researchers to think that a process model, as would appear a time-sharing operating system, was a perfect metaphor to reuse in the design of real-time computer music control systems.

Both the Samson box and the 4C maintained lists of parameter updates that resemble Music N scores in that they have sequences of numeric updates for synthesis parameters. In the case of the 4C, the scheduling of the updates could depend on real-time inputs. Both these systems, but particularly 4CED, modeled musical sequences in ways that resembled processes in the computer science sense.

4 Processes in modern computer music environments

The four most widely-used computer music environments are probably Csound, Max, Supercollider, and Pd. (Since this is a linux conference, we won't consider Max here, but only the closely related Pd.) Of all these, Supercollider is unique in that it explicitly adopts a process-like model, which offers at least two advantages. First, it allows the user to "think" in processes in order to express the parallelism that is desirable for polyphony, for instance in voice banks or collections of sinusoids in additive synthesis.

Second, it allows parallelism in the signal processing engine so that multiprocessors can be exploited.

A newer environment, ChucK[Wang and Cook, 2003], also uses a thread model and aims in part to make the creation and destruction of processes (named "shreds") as lightweight as possible. This language is still under active development and may lead to new ideas for adapting the concept of process to interactive computer music environments.

The process models in both Supercollider and ChucK both lend themselves well to generative applications, where processes may need to quickly and efficiently create new voices or instances of computational algorithms. In both environments the creation and destruction of processes is highly optimized so that large numbers of them may be created and managed dynamically.

Pure data, on the other hand, offers no model of a process and therefore is badly adapted both to expressing polyphony (although this is fixable using a voice bank management object available in Pd extended but not yet in "vanilla"). It is even less well adapted to expressing generative algorithms in which data may fork and recombine in ways that Supercollider and ChucK make easy. It is perhaps the most distinguishing feature of Max and later Pd that they both radically did away with the notion of process altogether.

Pd offers no easy way to manage parallelism either; the facility provided is the "`pd~`" object which can be considered a throwback to the Max/FTS solution from 1990. There are advantages gained by this trade-off. Most importantly (in my view at least), the fluency and ease by which Pd patches can react to input from the outside world is much greater. This is partly because of the absence of a process model, because one never has to consider how different processes must be synchronized in order to react consistently to new and possibly unpredictable inputs.

5 Conclusion

The notions of "process" and "thread" seem eternally attractive to designers of real-time computer music programming environments such as the ones discussed here. The attraction sees to be for both expressive reasons (as a way to describe polyphony, particularly in generative situations) and for efficiency reasons (as ways to efficiently exploit parallelism in general-purpose processors). Yet the difficulties of managing coordination between processes or threads still make it appear impossible to adapt them easily to an environment like Pd. This is a hard problem that is worthy of future work.

Meanwhile, the most powerful arithmetic processors in modern devices are their graphics processors. We don't yet have a good understanding of how to exploit these architectures for computer audio, and indeed this seems so far from today's programming models that it is hard to see where we could start on this.

It seems that the state of the art in programming environments for doing interactive computer music is out of sync with current developments in computing. Past efforts to make music out of computer hardware and operating systems that were often ill suited to the task have often resulted in advances that had implications not only for computer musicians but for computer science as well. Attacking the current situation in a similar way might similarly give rise to useful new ideas.

6 Acknowledgements

Thanks as always to the hard-working organizers of LAC for making this publication possible. This paper is based in part on material from a manuscript for a planned future book.

References

C. Abbott. 1981. The 4ced program. *Computer Music Journal*, pages 13–33.

D. Gareth Loy. 1981. Notes on the implementation of musbox: A compiler for the systems concepts digital synthesizer. pages 333–349.

Max V. Mathews. 1969. *The Technology of Computer Music.* MIT Press, Cambridge, Massachusetts.

J.A. Moorer, A. Chauveau, C. Abbott, P. Eastty, and J. Lawson. 1979. The 4c machine. *Computer Music Journal*, pages 16–24.

Ge Wang and Perry R. Cook. 2003. Chuck: A concurrent, on-the-fly audio programming language. pages 217–225, Ann Arbor. International Computer Music Association.

FAUSTLIVE
Just-In-Time Faust Compiler... and much more

Sarah DENOUX and **Stephane LETZ** and **Yann ORLAREY** and **Dominique FOBER**
GRAME
11 Cours de Verdun (GENSOUL)
69002 Lyon
FRANCE
{sdenoux, letz, orlarey, fober}@grame.fr

Abstract

FaustLive is a standalone just-in-time FAUST compiler. It tries to bring together the convenience of a standalone interpreted language with the efficiency of a compiled language. Based on *libfaust*, a library that provides a full in-memory compilation chain, FaustLive doesn't require any external tool (compiler, linker, etc.) to translate FAUST source code into binary executable code.

Thanks to this technology, FaustLive provides several advanced features. For example it is possible, while a FAUST application is running, to modify its behavior on-the-fly without any sound interruption. It is also possible to migrate a running application from one machine to another, etc.

Keywords

Audio, FAUST, DSP programming, remote processing and interfacing

1 Introduction

FAUST [Functional Audio Stream] [6] is a functional, synchronous, domain-specific programming language specifically designed for real-time signal processing and synthesis. A unique feature of FAUST, compared to other existing music languages like Max, PD, Supercollider, etc., is that programs are not interpreted, but fully compiled. FAUST provides a high-level alternative to C/C++ to implement efficient sample-level DSP algorithms.

But, if compilers have the advantage of efficiency, they have their own drawbacks compared to interpreters. Compilers traditionally require a whole chain of tools to be installed (compiler, linker, development libraries, etc.). For non-programmers this task can be complex. The development cycle, from the edition of the source code to a running application, is much longer with a compiler than with an interpreter. This can be a problem in creative situations where quick experimentation is essential. Moreover, binary code is usually not compatible across platforms and operating systems.

FaustLive is an attempt to bring together the convenience of a standalone interpreted language with the efficiency of a compiled language. Based on *libfaust*, a library that provides a full in-memory compilation chain, FaustLive is a standalone application that doesn't require any external tool to translate FAUST source code into binary executable code and run it. In many aspects FaustLive behaves like a FAUST interpreter with a very short development cycle (not very different, in that aspect, from modern compiled LISP environments, or from the approach presented by Albert Graef with Pure in [1]).

Moreover, FaustLive provides some advanced features to speedup the development cycle. For example, while a FAUST application is running, it is possible to edit and recompile its FAUST code on-the-fly, without any sound interruption. If the application is using JACK as driver, all audio connections are maintained. Another interesting feature is the possibility to migrate a running application from one machine to another through the network even across operating systems. Applications can also be controlled remotely, using HTTP or OSC.

FaustLive offers a lot of flexibility to prototype audio applications. It can also be connected to FaustWeb, a remote compilation service to export the application as a traditional binary for one of the various operating system and audio architecture supported by the FAUST ecosystem.

Since FaustLive is based on *libfaust*, the FAUST compiler project will first be presented (see Section 2). Then FaustLive will be shortly described through a typical use case (see Section 3) to finally be detailed over its technical aspects (see Section 4).

2 Faust Compiler

The FAUST compiler translates a FAUST program into an equivalent imperative program (typically C, C++, Java, etc.), taking care of generating efficient code. The FAUST package also includes various architecture files, providing the glue between the generated code and the external world (audio drivers and user interfaces).

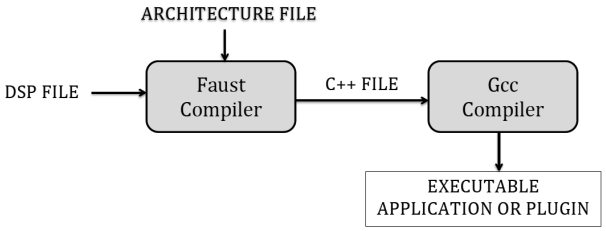


Figure 1: Steps of FAUST compilation chain

The current version of the FAUST compiler produces the resulting DSP code as a C++ class, to be inserted in the architecture file. The resulting C++ file is finally compiled with a regular C++ compiler to produce the final executable program or plug-in (Figure 1).

The resulting application is structured as shown in Figure 2. The DSP has become an audio computation module. As for the architecture, it turned into links to the user interface and the audio driver.

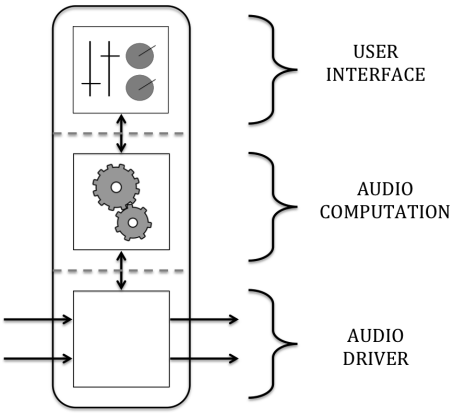


Figure 2: FAUST application structure

2.1 LLVM

LLVM (formerly Low Level Virtual Machine) is a compiler infrastructure, designed for compile-time, link-time, run-time optimization of programs written in arbitrary programming languages. Executable code is dynamically produced using a "Just In Time" compiler from a specific code representation, called LLVM IR[1]. Clang, the "LLVM native" C/C++/Objective-C compiler is a front-end for LLVM Compiler. It can, for instance, convert a C or C++ source file into LLVM IR code (Figure 3).

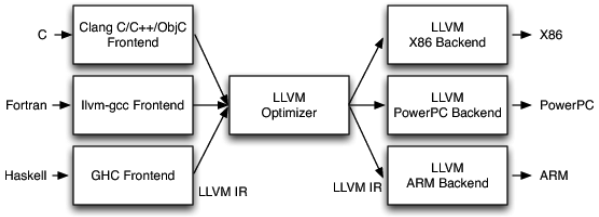


Figure 3: LLVM compiler structure

Domain-specific languages like FAUST can easily target the LLVM IR. This has been done by developing a special LLVM IR backend in the FAUST compiler, [5].

2.2 Dynamic compilation chain

The complete chain goes from the DSP source code, compiled in LLVM IR using the LLVM backend, to finally produce the executable code using the LLVM JIT. All steps are done in memory. Pointers on executable functions can be retrieved in the resulting LLVM module, and their code directly called with the appropriate parameters.

In the *faust2* development branch, the FAUST compiler has been packaged as an embeddable library called *libfaust*, published with an associated API, [2]. This API imitates the concept of oriented-object languages, like C++. The step of compilation, usually executed by gcc, is accessed through the function *createDSP-Factory*. Given a FAUST source code (as a file or a string), the compilation chain (FAUST + LLVM JIT) generates the "prototype" of the class, called llvm-dsp-factory. Then, the function *createDSPInstance*, corresponding to the "new className" of C++, instantiates a llvm-dsp. It can then be used as any object, run and be controlled through its interface.

Embedding this technology in a program or a plug-in enables dynamic modifications of the audio computation module of a FAUST application [4].

[1] *The Intermediate Representation* is an intermediate SSA representation

3 FaustLive - Use Case

FaustLive is a QT-based[2] software that permits to launch FAUST applications from their source code without having to precompile them (Figure 4).

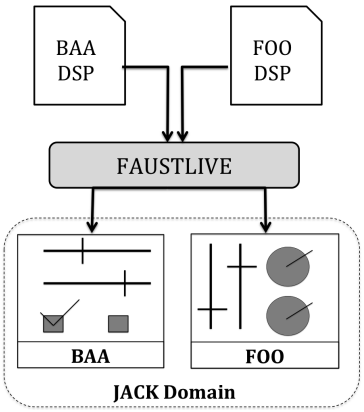


Figure 4: FaustLive principle

FaustLive exploit dynamic compilation, associated with multiple interfacing systems and audio drivers to modulate the structure of FAUST applications and simplify FAUST prototyping process.
To give an idea of FaustLive's potential, the following section presents its diversified features, showing the corresponding alterations in the structure of the applications.

The starting point of FaustLive's features is drag and drop. A FAUST DSP can be opened in a new window or it can be dropped on a running FAUST application. As a result, an intermediate state emerges in which the two applications coexist. The arriving application copies the established audio connections. Then, the output of the old application is cross-faded to the new one (Figure 5). At last, the dropped application durably replaces the previous one. With that system, a running application can be changed endlessly, without audio click.

This mechanism also allows source edition. When the user chooses to edit its FAUST code, it is opened in a text editor. And as his changes are saved, the application is updated using the crossfade mechanism (Figure 6).

[2] *QT* is a framework for interface design

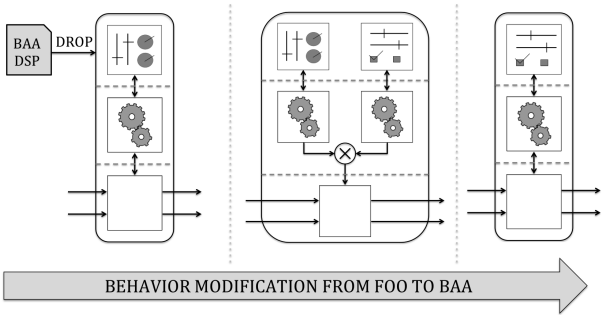


Figure 5: Behavior modification

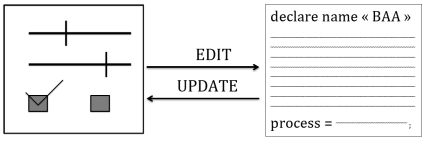


Figure 6: Dynamic source edition

JACK was primitively adopted as audio driver for it allows the user to connect its FAUST applications between themselves. Other drivers have then been added, making this component of the structure as flexible as the others. So when FAUST applications are running, FaustLive gives the possibility to dynamically switch the audio driver. FaustLive does not need to be stopped. The migration is made during execution and is applied to every FAUST application running. JACK, NetJack, CoreAudio and PortAudio are the integrated drivers in FaustLive (Figure 7).

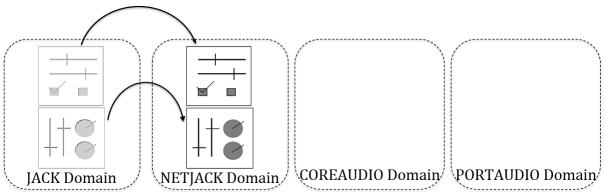


Figure 7: Dynamic driver migration

FaustLive expands its radius of action to external interactions. A smartphone can open an OSC[3] interface, controlling the application remotely (Figure 8).

Likewise, a HTML interface is accessible through a Qr Code[4]. By scanning it with a

[3] *OSC* : Open Sound Control

[4] *QR code* (abbreviated from Quick Response Code) is the trademark for a type of matrix barcode (or two-

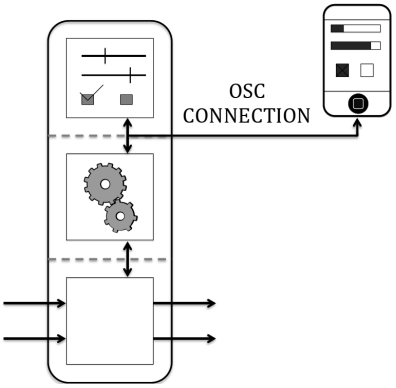


Figure 8: OSC interface

touchpad (for instance), the remote interface is opened in a browser. In both cases, the interface is duplicated and a synchronization between the local and remote interface is established.

The HTML interface has an additional interest: it is set up to enable drag and drop. Therefore, the user controlling the remote interface can also change the behavior of the application by dropping his own DSP. It is sent to the local application where it replaces the running one, using the crossfade mechanism. Finally, the remote interface is updated (Figure 9).

If many or/and heavy FAUST applications are opened, local CPU load can be saturated. The migration of calculations to other machines can lighten this load. On account of dynamic compilation, the audio computation module can be relocated on another machine (Figure 10). The list of remote servers available is built dynamically so that it is simple to switch from local processing to remote processing.

A user may wish to run his FAUST application in an other environment (Max/MSP, SuperCollider, ...). For that matter, a link to FaustWeb, a remote compilation web service, is integrated in FaustLive. The user only has to choose the platform and environment he wishes to target. In return, he will receive the binary of the requested application or plugin.

When FaustLive is exited, the last configuration is saved and will be restored at its next

dimensional barcode)

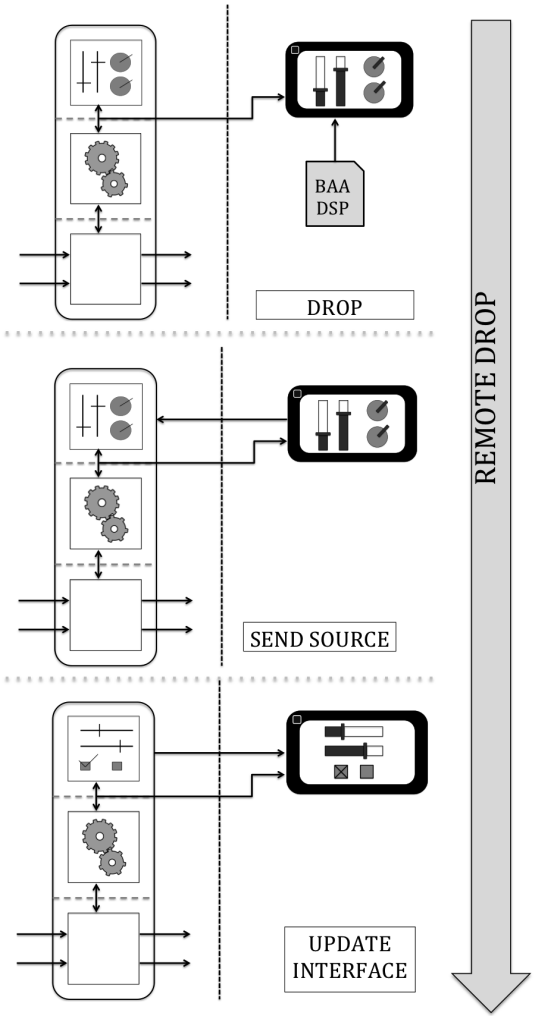


Figure 9: Remote drop

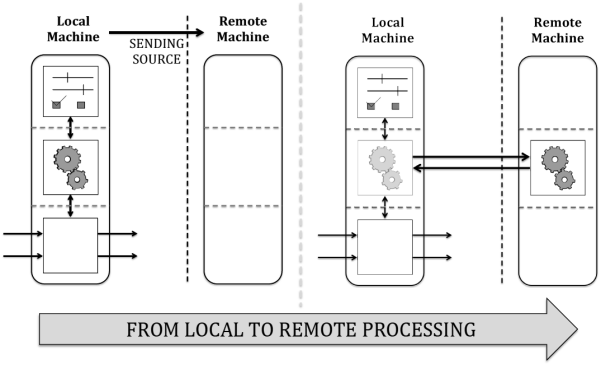


Figure 10: Remote processing

execution. A user may also save the state of the application at any moment. In a second phase, he will be able to reload his snapshot, by importing it in the current state or recalling it (Figure 11).

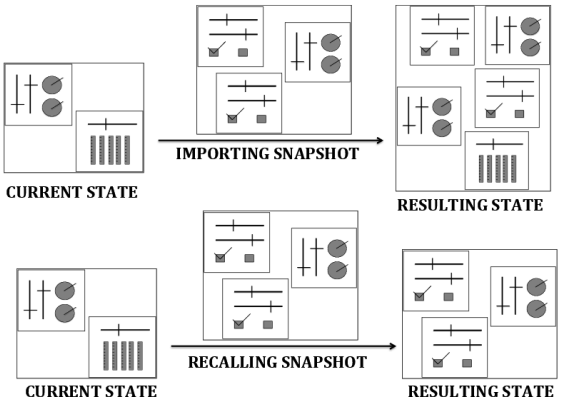


Figure 11: Reloading snapshot

4 FaustLive - Technical View

4.1 Basic FaustLive Features

The first aim of FaustLive is to create a dynamic environment for FAUST prototyping, by embedding *libfaust*. The resulting dynamic compilation chain (Figure 12) presents the advantage of speeding up the compilation process. Returning almost right away the executed application, this compiler is a stepping stone for dynamic behaviors.

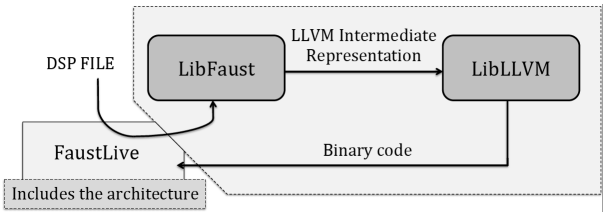


Figure 12: Compilation chain in FaustLive

Now that it is possible to dynamically compile FAUST code, new prospects are rising. A user may drop his FAUST code as a file, a string or a url, on a running application. As a result, the code is immediately given to the embedded compiler and the new application replaces the previous one. Since FaustLive is designed for dynamic uses, it is very important to ensure a continuity in the sound. For that matter, a crossfade is calculated between the two relaying FAUST applications.

Moreover, a FAUST application is linked to its source, so that any modification in the FAUST code will lead to a recompilation. This particular aspect is central, for it simplifies the prototyping process: a user can modify his code at leisure and see/hear instantly the result.

An important asset of FaustLive is the coexistence of multiple FAUST applications, in opposition with the QT-JACK architecture from FAUST "static" distribution, where every FAUST program has to be compiled separately to produce its own application. Here, each application evolves with the actions it undergoes and has its own set of dynamic parameters (Figure 13).

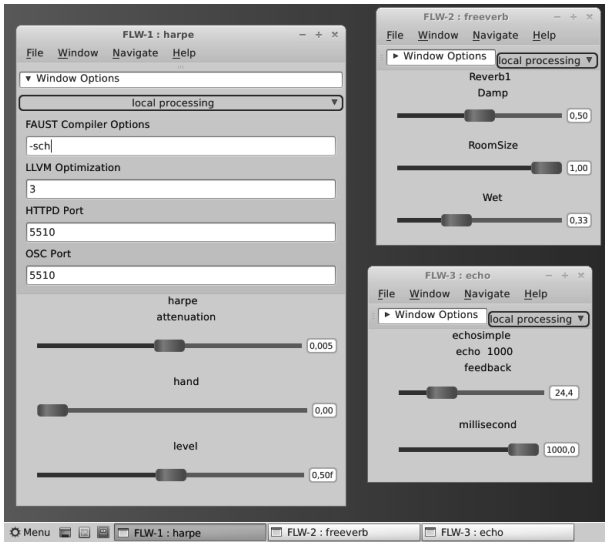


Figure 13: FaustLive's environment

4.2 Audio Drivers

FaustLive has integrated JACK, CoreAudio, NetJack and PortAudio[5]. So that it's possible to switch audio structures or modify its parameters (such as buffer size or sample rate) during FaustLive's execution. Every running audio client is stopped, then the applications are transferred in the new domain to finally be restarted.

4.2.1 JACK

JACK is a system for handling real-time, low latency audio (and MIDI). It runs on GNU/Linux, Solaris, FreeBSD, OS X and Windows. It can connect a number of different applications to an audio device, as well as allowing them to share audio between themselves.

[5]JACK, CoreAudio and NetJack are used on OSX, JACK and NetJack on Linux, PortAudio, JACK and NetJack on Windows.

Therefore, an interesting constraint in using JACK is the matter of the connections. When connections have been established, the objective is to maintain them even if the FAUST application changes in a window. If the new application has more ports than the previous one, the user will have to make the connections himself.

4.2.2 NetJack

NetJack is a Realtime Audio Transport over a generic IP Network, fully integrated into JACK. NetJack synchronizes all clients to one soundcard, so there is no resampling or glitches in the whole network. The master imposes the sample rate and buffer size, in relation to its audio device.

4.2.3 CoreAudio and PortAudio

Because the protocol has to be strictly the same on the client and on the server's side, JACK and NetJack have to be linked as a dynamic library. The problem it brings is that FaustLive's installation is linked to JACK's installation. To avoid this inconvenience for beginner users, a CoreAudio[6] and PortAudio[7] versions have been developed. Included in the standard libraries or easily linked as a dll, they do not expand the user's work.

4.3 Control Interfaces

To offer a modular application, FaustLive expands the choices of the user, concerning the control interface.

4.3.1 OSC Interface

OSC protocol is integrated to FaustLive to offer another type of interface and enable interoperability. Many audio environments and devices implement this protocol so that FaustLive will be able to communicate with them. The user can configure the port on which the protocol is started and then control the interface with, for instance, an OSC touch application.

[6] *CoreAudio* is the digital audio infrastructure of iOS and OS X. It provides a framework designed to handle audio needs in applications.

[7] *PortAudio* is a free, cross-platform, open-source, C/C++ audio I/O library. It is intended to promote the exchange of audio software between developers on different platforms.

4.3.2 HTML Interface

FAUST HTML interface is also a component of FaustLive. Loaded on any browser, this interface controls the DSP's parameters, through a HTTP connection. When it is built, a server is started, taking care of delivering the HTML page (Figure 14). A synchronization between the local and the remote interface is also insured.

To ease the opening of the interface, a Qr Code is built from the HTTP address, thanks to *libqrencode*. Most smartphones and portable equipments have a QrCode decoder. By scanning the Qr Code, a browser gets connected to the interface page.

4.3.3 Preferences

The challenge FaustLive was confronted with is to provide an interface that gives as many liberties as possible to the user all the while being easy to apprehend. In that direction, OSC and HTTP ports are configurable in the window's options. The window toolbar is collapsed, by default, so that a "basic" user won't feel assailed by preferences (Figure 13). Both protocols use 5510 as default port. When the TCP listening port number is busy, the system automatically looks for the next available port number.

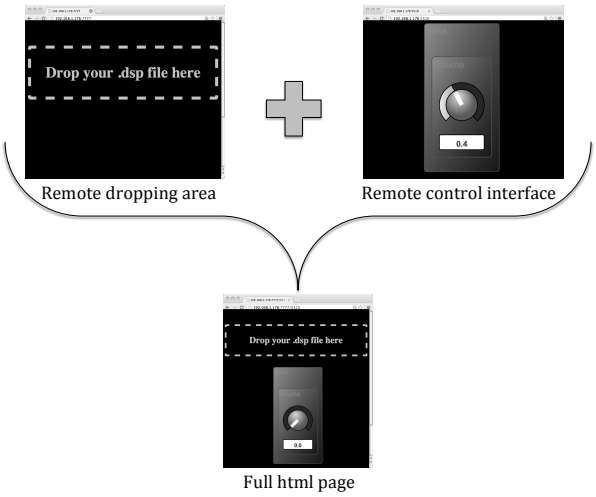


Figure 14: HTML interface with control and dropping services

4.3.4 Remote Drag and Drop

As the rest of FAUST current distribution, the HTML interface has a "static" behavior.

The intention, to copy FaustLive's dynamic behavior, led to adding a dropping area to the HTML interface. This HTTP service is independent and specific to FaustLive. The server, started by FaustLive, is able to create a HTML page that encapsulates the remote interfaces. The resulting service of remote interface and DSP drop has the following address: http://IP:DroppingPort/InterfacePort (Figure 14).

The dropping port is set in the preferences and is common to all the FAUST applications. The remote interface port is distinct for every FAUST application and editable in the window's options.

The reaction to the drop follows FaustLive's model. The DSP is first sent to FaustLive as a HTTP post request. The DSP is compiled and replaces the previous one, after the crossfade. At last, the remote interface is updated.

4.4 Remote Processing

To widen its benefits, FaustLive enables remote processing. The compilation and process calculation are redirected on a remote machine and local CPU load can be lightened.

On a remote machine, an application starts a HTTP server, offering the remote compilation/processing service. This server is waiting for requests.

On the client's side (FaustLive), an API "proxy" makes it transparent to create a remote-dsp rather than a local llvm-dsp (c.f 2.2). This API, *libfaustremote*, takes care of establishing the connection with the server.

The first step (compilation) is carried out by the function *createRemoteDSPFactory*. The code is sent to the server, which compiles it and creates the "real" llvm-dsp-factory. The remote-dsp-factory returned to the user is an image of the "real" factory. Before sending the FAUST code, a FAUST to FAUST compilation step is executed locally, to solve all the dependencies. This way, the expanded code sent to the server is self-contained.

The remote-dsp-factory can then be instantiated to create remote-dsp instances, which may run in the audio/visual architecture chosen (here, FaustLive).

To be able to locally create the interface, the server returns a json-encoded interface. This way, the function *buildUserInterface* can be recreated, giving the impression that a remote-dsp works as a local llvm-dsp.

Moreover, the audio processing is redirected through a NetJack connection. The audio data is sent to the remote machine which processes it and sends back its results. In addition to the standard audio flow, one midi port is used to transfer the controllers values (Figure 15). The benefit of this solution is to transmit synchronized audio and controllers in the same connection. Moreover, the audio samples can be encoded using the different possible audio data types : float, integer, and compressed audio (using the OPUS codec[8]).

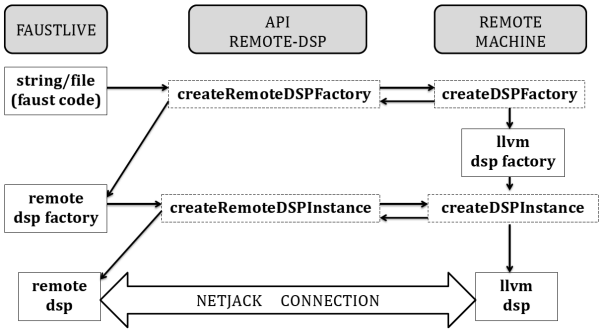


Figure 15: Remote compilation

libfaustremote uses *libcurl* to send http requests to the remote server, handled with *libmicrohttpd*.

On FaustLive's windows, the service of remote processing is simply interfaced. The Zero-Conf protocol is used to scan the remote machines presenting the service. A list is then dynamically built with the available ones. By browsing in the list, the user can then switch from a machine to another or come back to local processing very easily.

4.5 FaustWeb

In order to simplify the accessibility of the FAUST compilation, this web service of remote compilation has been conceived. It receives a FAUST DSP and returns a plugin or application in the chosen target architecture. As an outcome, the installation of FAUST package and all additional SDKs on the user machine is not necessary anymore. Anyone can write a FAUST

[8]http://www.opus-codec.org

application, send it to the server and receive a plugin.

This service is accessible from a browser but requires several requests. Through FaustLive, the export is facilitated. A menu is dynamically built with the platforms and architectures available. And as the user makes his choice, his code is sent to the server. The first step is the syntax verification, returning a sha1 key, with which multiple requests can be made. The second step is the compilation, using the standard "static" chain and returning the chosen application to the user (Figure 16).

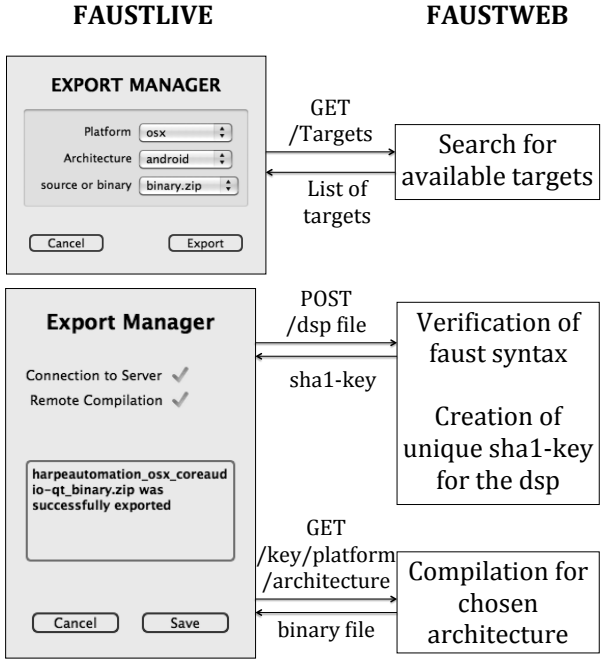


Figure 16: Steps of the compilation chain through FaustWeb

4.6 Session Management

A concept of session is introduced to preserve the state of the application (parameters values, position on screen, audio connections, compilation options, ...) when the application is closed or when the user takes a snapshot, which saves his session in a .tar file.

A FaustLive snapshot is self-contained. All the local resources needed (like FAUST DSPs) are copied into the folder. Pointers to the resources are used as much as possible. But if a source file is erased or the snapshot is transferred on another machine, copies ought to be employed.

To decrease the compilation time, the output of FAUST compiler, the optimized LLVM IR code, is saved. When the application is recalled, FAUST compiler's and LLVM IR to IR optimization steps are skipped. For very heavy programs, the gain can be noticeable (from a few seconds to almost instantaneous).

5 Conclusion

FaustLive brings together the convenience of a standalone interpreted language with the efficiency of a compiled language.

FaustLive offers currently the shortest development cycle for FAUST applications, allowing even to modify the code of an application while it is running. It integrates advanced remote computation and control features for real-time distributed audio applications. Moreover FaustLive provides, via its export functionality, a convenient front-end for FaustWeb, the compilation web service of FAUST. The project is open-source and available on Sourceforge [3]. It runs on Linux, OSX and Windows.

Acknowledgments

This work has been implemented for one part under the INEDIT project [ANR-12-CORD-0009] and for the other part under the FEVER project [ANR-13-BS02-0008]. It is supported by the "Agence Nationale pour la Recherche" .

References

[1] A. Graef. Functional signal processing with pure and faust using the llvm toolkit. 2011.

[2] faust2 repository. http://sourceforge.net/p/faudiostream/code/ci/faust2/tree/.

[3] faustlive repository. http://sourceforge.net/p/faudiostream/faustlive/ci/master/tree.

[4] S. Letz, Y. Orlarey, and D Fober. Comment embarquer le compilateur faust dans vos applications? 2013.

[5] S. Letz, Y. Orlarey, and D Fober. Dynamic compilation of parallel audio applications. 2013.

[6] Y. Orlarey, S. Letz, and D. Fober. Faust: an efficient functional approach to dsp programming. 2009.

OpenMusic on Linux

Anders VINJAR*
Composer
Norway
anders.vinjar@bek.no

Jean BRESSON
STMS lab
IRCAM-CNRS-UPMC
1, place Igor Stravinsky
75004 Paris, France
jean.bresson@ircam.fr

Abstract

We present a recent port of the OpenMusic computer-aided composition environment to Linux. The text gives a brief presentation of OpenMusic and typical use-cases of the environment. We also present a short history of its development, and mention previous attempts at porting it to Linux. The main technical challenges involved with developing the current Linux port are discussed, as well as solutions to these. We end the paper by outlining some possible areas for future work.

Keywords

OpenMusic, Computer-Aided Composition (CAC), Linux, Common Lisp, Visual Programming, JACK.

1 Introduction

OpenMusic (OM) is an environment for computer-aided music composition designed as a domain-specific visual programming language.[1] It allows composers to write and run programs to transform or generate data, with interactive access to the input or output musical structures.

Before 2013 OM was developed and distributed on OS X and Windows platforms only, despite various attempts at porting the environment to Linux. In this paper we present a new, fully functional port of OM to Linux.

In Section 2 we will introduce the OpenMusic environment, first from a general point of view, and then discuss some particular aspects such as the user interface and the external dependencies of the environment. We also give a quick history of the development of OM, and previous attempts at porting the environment to Linux.

Section 3 describes our current implementation choices and the state of the Linux port. We conclude with a number of perspectives and areas for future work.

* This work is supported by BEK - Bergen Center for Electronic Arts.

[1] http://repmus.ircam.fr/openmusic/

2 OpenMusic

2.1 A visual programming environment for computer-aided composition

OM is a visual programming environment dedicated to music processing and composition [Assayag et al., 1999]. It implements the main features of the Common Lisp language (abstraction, higher-order functions, recursion, iterations etc., see [Bresson et al., 2009]),[2] as well as object-oriented programming [Agon and Assayag, 2003] and constraints programming [Rueda et al., 1998].

OM is primarily an environment for work in computer-aided composition (CAC). It is also used for musicological tasks like analysis, modeling or statistics, as well as pedagogical work in composition studies or music theory [Bresson et al., 2011]. The environment comes with a rich set of tools and libraries aimed at composition, analysis, DSP and other musical/extra-musical domains.

The aim of a CAC application is to aid the user in typical composition tasks like generating, representing and manipulating musical material in adequate ways, handling musical form as material develops towards a finished piece of music. Contemporary composition is a rather ill-defined activity,[3] and a good CAC tool is one with abilities to adapt well to human artistic processes, whatever those may be. Indeed, composers seldom follow strict plans for too long. More common is perhaps making up a class of material, generate versions on that, develop this again further towards some more complex structures, go back to change some-

[2] Functional programming and Lisp in particular has a strong background and tradition in the computer music history. It has proven to be relatively easy to learn, understand and use by composers.

[3] *"CAC is in effect making the computer carry out thought processes previously carried out in human brains"* (Miller Puckette, preface to *The OM Composer's Book vol. 1* [Puckette, 2006]).

thing in an earlier stage, to arrive at yet another set of variants, and so on.

Interactive (visual) programming languages are well suited environments for composers to work efficiently and creatively with computers. The graphical environment in OM provides an interactive patch-based, data-flow approach, making it easy and intuitive to get going. Figure 1 shows a basic patch window generating a series of chords.

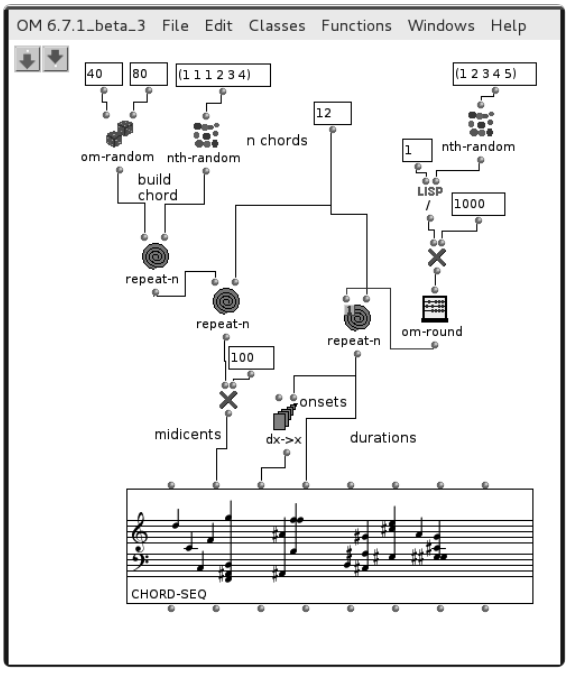


Figure 1: Graphical programming with OM.

OMs dynamic environment supports the kinds of interactive work-flows often preferred by composers, by allowing the user to evaluate and store output at arbitrary stages in the program-flow, and easy stash away variants for any kind of musical material or data.

OM is also a full-blown environment for Lisp-programming, with access to the expected REPL[4], special editors for Lisp code, cross-referencing and look up in source-code, and interactive help-system. These tools provide an environment for composers and programmers to develop new ideas and specialized creative work-flows, using graphical programming, textual programming, or any combination of these. In OM there is no particular separation between a function defined as an abstraction in a patch box or one defined by evaluating Lisp code: both are accessible as graphical objects in the user-interface.

[4]REPL = Read-Eval-Print loop.

Reports of some composition tasks and approaches carried out with OpenMusic are available in [Agon et al., 2006 2008].

2.2 User Interface

As a visual programming environment, OM is highly concerned with graphical user interfaces (GUI). To a large extent, this aspect determines the choices of platforms and frameworks that may be used for development.

While programming in OM, the user inserts and manipulates graphical objects in patch windows, dragging connection-lines between inlets and outlets of boxes, and thus establishing the data-flow of a musical program. These graphical objects may represent simple operators, abstractions or sub-patches, or more complex data like lists, arrays, break-point functions etc.

As part of the graphical programming environment, OM provides advanced editors for visualization and manipulation of data such as musical scores, break-point functions, audio waveforms and some other data types. Figure 2 shows the *sound*-editor, and Figure 3 shows editors for various other kinds of musical data.

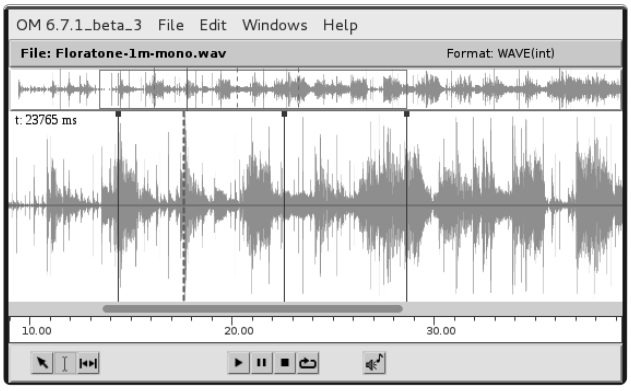


Figure 2: Editor for the OM `sound` object.

OM also has a programmable graphical timeline editor termed the "Maquette", where everything else which lives in OM – functions, musical data, connections – can be placed and manipulated, either manually or as a result of evaluating some code.

2.3 Development history – Previous ports and platforms

OpenMusic is a descendant of the Patchwork environment [Laurson and Duthen, 1989], one of the pioneering visual programming systems dedicated to computer-aided composition.

The first release (C. Agon and G. Assayag, IRCAM) was built in 1998 on Mac OS using

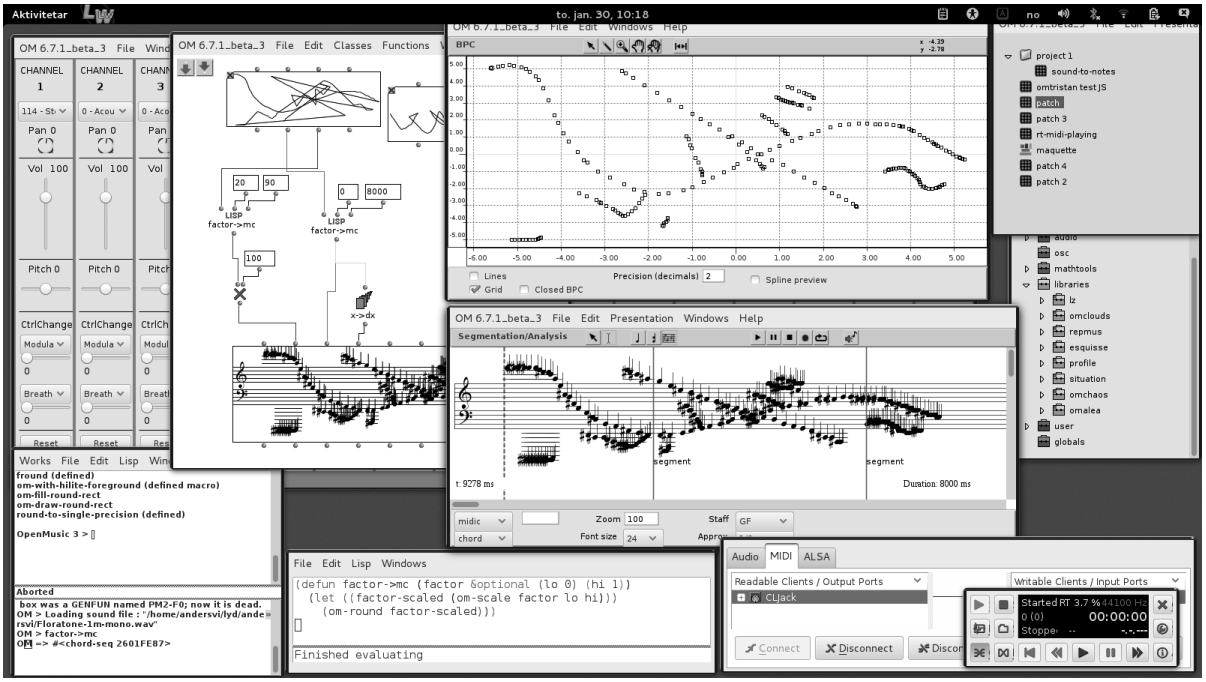


Figure 3: OM environment: musical data and editors.

Digitools' Macintosh Common Lisp (MCL). The graphical programming system was designed as a full meta-programming framework, implementing functional programming concepts and interactions, on a strong base of object-oriented programming and CLOS (Common Lisp Object System [Gabriel et al., 1991]). OM also introduced a number of new concepts concerning for instance handling and manipulation of objects, or unified representation of time structures.

In 2003, OM 4 was ported to OS X, and a first Linux port [Sarria and Diago, 2003] was carried out at IRCAM in the framework of the AGNULA European project [Déchelle and Tisserand, 2003], using CMUCL[5] and Gtk+[6].

Released in 2005, OM 5 [Bresson et al., 2005] was a multi-platform version of OM developed on Mac (using MCL) and on Windows (using Allegro CL[7]). The OM 5 sources were clearly divided into a platform-independent kernel built on top of an abstract graphical/system-dependent API inspired by the MCL toolkit (MCL and ACL versions of this API were then implemented and loaded depending on the targeted platform).

A second port of OM on Linux was initiated in 2006 based on the OM 5 sources, using SBCL[8] and a new implementation of the OM API for Gtk+ and CLG[9] (Common Lisp GTK bindings). Unfortunately this project was never carried to its end.

In 2006, Digitools announced discontinuation of MCL development and support on Mac, due to the switch to Intel processors. The LispWorks environment was chosen as a replacement, providing a reliable IDE with a common cross-platform API (CAPI) compatible with the main graphic toolkits for Mac, Windows and Linux, as well as some other OS-es.

In 2008 OM 6 based on LispWorks was released for OS X and Windows.

2.4 External dependencies

OM is dependent on audio I/O systems due to its musical orientation. It is important for composers to be able to load audio or MIDI files, and convert/process the contained data in the visual programming framework. Playback and rendering of generated musical material (score/MIDI or audio) is an essential feature of the environment, and complex scheduling issues can arise when dealing with multiple simultaneous sources and audio/MIDI material interactively. Still, OM is not by nature a real-time environment and the audio performance requirements remain relatively moderate

[5] http://www.cons.org/cmucl/
[6] http://www.gtk.org/
[7] http://www.franz.com/products/allegrocl/
[8] http://www.sbcl.org/
[9] http://sourceforge.net/projects/clg/

as compared to real-time scheduling or audio processing systems.

Low-level MIDI formatting and scheduling was traditionally supported in OM by the MidiShare[10] system [Orlarey and Lequay, 1989]. The audio support, initially limited to a few functions interfacing with the Apple QuickTime library, was replaced in OM 5 by a more advanced and multi-platform audio support developed on top of the LibAudioStream[11] library.

Besides MIDI for file I/O, support for exporting other score-type data is provided either by using built-in code or by using external libraries. OM can export to some much used file formats for sheet-music, like LilyPond[12] and MusicXML[13]. OM also features support for connections to technologies like SDIF [Schwartz and Wright, 2000] (standard format for interchange of sound description data), OpenGL (display of 3D objects in OM editors) and OSC [Wright, 2005] for inter-application communication, although none of these are strictly required to get the visual programming environment running.

OM communicates with external libraries via Common Lisp foreign function interfaces (FFI). OM either uses the FFI provided by the Lisp implementation, or the CFFI system [Bielman and Oliveira, 2013], a common FFI wrapper compatible with several Lisp implementations.

2.5 Distribution and licensing

While the first OM releases were distributed commercially along with other IRCAM software, since OM 6.4 (2011), the compiled OM application has been available free of charge for all platforms.

The source-code has always been freely available under the GNU Public License.

All source-code and external libraries required for building OM are open-source. However, to build and save the actual distributed image a LispWorks Professional or Enterprise license is necessary at this moment.

3 Towards OM on Linux

Several previous efforts to port OM to Linux over the years suggest a real interest, and OMs main technological dependency (a professional ANSI Common Lisp implementation with interfaces to graphical toolkits, MIDI and audio libraries) has been available to Linux developers for long. Hence, it is legitimate to ask why these previous ports did not succeed too well?

First, the Musical Representations team developing OM at IRCAM use Mac OSX as main development platform. As a consequence, porting OM to the attempted environments (CMUCL or SBCL w. Gtk) means rewriting all the graphical dependencies using foreign toolkits or alternative APIs.

Audio and MIDI support has also been an issue in previous Linux ports. Although MidiShare worked fine with earlier Linux-versions,[14] it is no longer maintained and kept compatible with newer releases of the Linux kernel. Moreover, OM MIDI dependencies were since the early releases packed together and integrated with the kernel code, making it difficult to replace.

As OM has evolved, work towards a more modular structure has been an explicit aim, at least since 2005 [Bresson et al., 2005]. Recent developments have gradually made the code more durable and resistant towards changes in compiler-implementations. This tendency both motivated and greatly helped the work with the current port, making it possible to develop alternative solutions for e.g. MIDI I/O, separated from those for the kernel, graphics or audio I/O.

3.1 OM 6.8 on Linux: Current State

When starting this project, a separate Linux development-branch of the source-tree for OM 6.7 was set up, making it possible to get up to speed on Linux without halting further development on the main-branch. At a certain stage the separate Linux-branch was re-integrated with the main source-code. Further development of the application, and delivery of images, is now based on the same source-tree for all three supported platforms – Linux, OS X, Windows – with only a few specializations, mainly in the graphics-code, to account for differences across platforms.

The present Linux development is based on OM 6.8 and LispWorks 6.1. The choice of LispWorks (a commercial Lisp compiler and IDE) for porting OM to Linux is entirely pragmatic: as mentioned, LispWorks provides a common API across graphical toolkits (Gtk+, Motif, Cocoa, Windows), and since this library is

[10] http://midishare.sourceforge.net/
[11] http://libaudiostream.sourceforge.net/
[12] http://www.lilypond.org/
[13] http://www.musicxml.com/

[14] MidiShare was also used in e.g. Common Music [Taube, 1991].

already used by the OM developers, only moderate adaptions to the existing OM API are required in order to get it working with Linux.

As suggested, this port of OM to Linux can be seen as the most recent in a series of steps towards making OM more modular. Solutions which work across platforms are presumably also less vulnerable to changes in compilers or toolkits in use. While developing Linux-compatible substitutes for previous code, general and platform-independent solutions were looked for. In particular, most external dependencies have been made optional, so that OM can run without some specific features, if dependencies are not found, not loaded or not available for a specific environment or platform.

3.2 Audio and MIDI I/O: JACK

To substitute the low-level I/O systems for MIDI and audio, some alternative approaches were programmed and tested. The OM "player" system was rewritten in OM 6.7 as a modular API, making it easier to substitute or switch the default playback-engines with alternative audio or MIDI players.

MIDI messages and Standard MIDI Files can now be parsed and formatted using the Common Lisp MIDI library[15], and E. de Castro Lopo's `libsndfile` is used to access audio files on disk. The default playback-engines used in the Linux version of OM depend on `libjack`. Scheduling and real-time I/O of audio and MIDI between OM and hardware-ports (or between OM and other applications) have been programmed as a CL-based JACK[16] client.

Other test-case playback-engines were also developed, for instance with SuperCollider (Audio-MIDI I/O, scheduling) using OSC communication, with a CL-based FluidSynth[17] server (MIDI) controlled through Lisp-code, and with external MPlayer-processes[18] (audio) controlled from a sub-shell. These clients work fairly well, and could be used as examples for users or developers wanting to plug in other playback-engines or I/O systems. An alternative audio player has been developed for instance using `sox`[19] as part of the *OM-Sox* library.[20]

[15] http://www.doc.gold.ac.uk/isms/lisp/midi/
[16] http://jackaudio.org/
[17] http://www.fluidsynth.org/
[18] http://www.mplayerhq.hu/
[19] Sound eXchange – sox.sourceforge.net/
[20] M. Schumacher, *OM-Sox*: http://sourceforge.net/projects/omsox/

3.3 Libraries

OM comes with a number of specialized tools and libraries aimed at composition, analysis, DSP and other musical tasks. Some of these are "official" libraries: either distributed and maintained as part of the main OM package, or used to integrate OM with external tools such as Csound or some of the DSP engines available from IRCAM (SuperVP, PM2, Spat, Diphone, Chant).[21] A rich set of 3rd-party libraries, maintained by individual composers or developers, are also available for download,[22] together with simple how-tos for adding external libraries (see Figure 4).

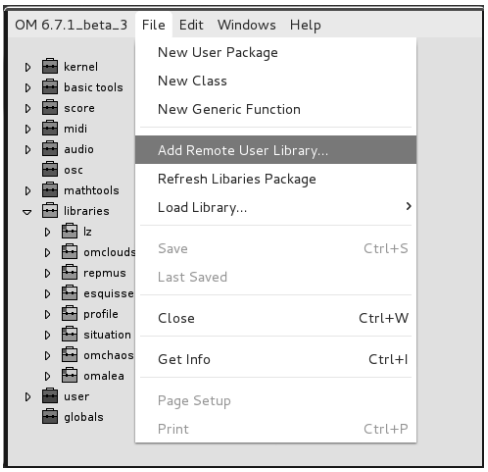


Figure 4: OM: Packages library.

Libraries are written as standard Lisp code, so they generally work well with the Linux-port. However, those depending on platform-specific external tools (e.g. OM-Spat) are not very useful at the moment.

3.4 Source code and packaging

The OM source code is distributed as part of the application.[23] The environment features runtime introspection and provides an easily accessible cross-reference for all OM-specific classes and methods. Lisp code may be edited and evaluated interactively, or loaded from files, and may be used to specialize or modify built-in functionality.

To compile a fresh OM from sources, access to LispWorks is necessary. For this reason OM is always made available as a pre-built image, one for each platform-type. At the time of this

[21] http://forumnet.ircam.fr/product/openmusic-libraries/
[22] http://repmus.ircam.fr/openmusic/libraries
[23] http://repmus.ircam.fr/openmusic/sources

writing, the Linux version is developed and maintained on a system running Fedora 19. Currently, OM is available as a `RPM`-package containing the image and all sources, which will install the binary and all sources in the usual places. Several users have adapted the RPM-packages to `dpkg`-based systems, seemingly without any serious issues, and also shared how-tos and experiences on the OpenMusic forums[24].

Patches are usually distributed as Lisp files. These can be dynamically loaded by the user or be automatically sourced on application start by placing the files in a predefined location.

4 Conclusions – Future works

The first beta-release of OM-Linux was made available for download and presented at the IRCAM Forum workshops in November 2013, after having been tested and used by developers and users for some time.[25]

The previous Linux ports missed their potential audience and lacked support and follow up by the Linux developers and composer's community, presumably due to a number of obstacles and difficulties that we have tried to outline in this paper. Our hope is that this project will overcome most of these problems.

A working Linux-version of OM is useful for end-users, and Linux-developers may well find good ways to integrate OM with other applications through e.g. libraries, or to develop OM further. The stabilization of the GUI API may also help to lessen dependencies on LispWorks in the future for all platforms, and make alternative open-source solutions possible. In this effort, starting out from a functional version on Linux may be valuable.

Since the current code uses one common source-tree, the Linux-port is relatively robust and sustainable across changes in compiler-implementations or frameworks for graphics and I/O. It also minimizes the work involved in maintaining compatibility across platforms for the same application.

The features introduced to make the sources compile and run on Linux, as well as the new developed support e.g. for audio and MIDI are of a general kind, and potentially useful across platforms. The CL-based MIDI-library, as well as the JACK-client and callback setup for MIDI and Audio, are now possible alternatives also for OS X and Windows.

Further work will be done in the near future to integrate other CL-based composition and DSP-tools such as Common Music, or Common Lisp Music (CLM[26]) [Schottstaedt, 1994], and to extend the existing set of libraries connecting OM with open-source software like e.g. SuperCollider or LilyPond. While the JACK-client is currently useful, real-time robustness can still be improved, and other audio-libraries or backends may also be supported in the future.

5 Acknowledgments

The authors would like to thank Trond Lossius and BEK (Bergen Center for Electronic Arts) for their support on this project. This work is also partly developed in the framework of the French ANR project with reference ANR-13-JS02-0004-01.

References

C. Agon and G. Assayag. 2003. OM: A Graphical Extension of CLOS using the MOP. In *Proc. of International Lisp Conference*, New York, USA.

C. Agon, G. Assayag, and J. Bresson, editors. 2006–2008. *The OM Composer's Book (2 volumes)*. Delatour/IRCAM.

G. Assayag, C. Rueda, M. Laurson, C. Agon, and O. Delerue. 1999. Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic. *Computer Music Journal*, 23(3).

J. Bielman and L. Oliveira. 2013. CFFI home page. http://common-lisp.net/project/cffi/.

J. Bresson, C. Agon, and G. Assayag. 2005. OpenMusic 5: A Cross-Platform Release of the Computer-Assisted Composition Environment. In *Proc. 10th Brazilian Symposium on Computer Music*, Belo Horizonte, MG, Brasil.

J. Bresson, C. Agon, and G. Assayag. 2009. Visual Lisp/CLOS Programming in OpenMusic. *Higher-Order and Symbolic Computation*, 22(1).

J. Bresson, C. Agon, and G. Assayag. 2011. OpenMusic – Visual Programming Environment for Music Composition, Analysis and Research. In *Proc. ACM MultiMedia*, Scottsdale, AZ, USA.

[24] http://forumnet.ircam.fr/user-groups/

[25] A review of this Linux port has been posted by D. Philips on LWN in November 2013, see https://lwn.net/Articles/574593/.

[26] https://ccrma.stanford.edu/software/clm/

F. Déchelle and P. Tisserand. 2003. The AGNULA project. Free software developments at IRCAM. In *International Workshop on Free Software and Research*, Compiègne, France.

R. P. Gabriel, J. L. White, and D. G. Bobrow. 1991. CLOS: Integration Object-oriented and Functional Programming. *Communications of the ACM*, 34(9).

M. Laurson and J. Duthen. 1989. Patchwork, a Graphic Language in PreForm. In *Proc. International Computer Music Conference*, Ohio State University, USA.

Y. Orlarey and H. Lequay. 1989. MidiShare : a Real Time multi-tasks software module for Midi applications. In *Proceedings of the International Computer Music Conference*, Ohio State University, USA.

M. Puckette. 2006. Preface. In C. Agon, G. Assayag, and J. Bresson, editors, *The OM Composer's Book .1.* Delatour/IRCAM.

C. Rueda, M. Laurson, G. Bloch, and G. Assayag. 1998. Integrating Constraint Programming in Visual Musical Composition Languages. In *Proc. European Conference on Artificial Intelligence*, Brighton, UK.

G. Sarria and J. F. Diago. 2003. OpenMusic for Linux and MacOS X. Technical report, IRCAM.

Bill Schottstaedt. 1994. Machine Tongues XVII: CLM: Music V meets Common Lisp. *Computer Music Journal*, 18(2):30–37.

D. Schwartz and M. Wright. 2000. Extensions and Applications of the SDIF Sound Description Interchange Format. In *Proc. International Computer Music Conference*, Berlin, Germany.

H. Taube. 1991. Common Music: A Music Composition Language in Common Lisp and CLOS. *Computer Music Journal*, 15(2).

M. Wright. 2005. Open Sound Control: An Enabling Technology for Musical Networking. *Organised Sound*, 10(3).

Radium: A Music Editor Inspired by the Music Tracker

Kjetil Matheussen
Norwegian Center for Technology in Music and the Arts. (NOTAM)
Sandakerveien 24D, Bygg F3
N-0473 Oslo
Norway
k.s.matheussen@notam02.no

Abstract

Radium is a new type of music editor inspired by the *music tracker*. Radium's interface differs from the classical music tracker interface by using graphical elements instead of text and by allowing musical events anywhere within a tracker line.

Chapter 1: The classical music tracker interface and how Radium differs from it. Chapter 2: Radium Features: *a*) The Editor; *b*) The Modular Mixer; *c*) Instruments and Audio Effects; *d*) Instrument Configuration; *e*) Common Music Notation. Chapter 3: Implementation details: *a*) Painting the Editor; *b*) Smooth Scrolling; *c*) Embedding Pure Data; *d*) Collecting Memory Garbage in C and C++. Chapter 4: Related software.

Keywords

Radium, Music Tracker, GUI, Pure Data, Graphics Programming.

1 Introduction

The tracker interface appeared on the AmigaOS platform in late 80s and early 90s with programs such as Soundtracker, NoiseTracker and Protracker. The first tracker was called "The Ultimate Soundtracker",[1] and was released in 1987 by Karsten Obarski.[2]

In the classical tracker interface, time goes downwards. Notes placed higher on the screen are played before notes placed below.[3] Instead of moving the cursor up or down, the whole editor scrolls up or down, and the cursor is just locked in the middle of the screen.

The tracker editor shows a two-dimensional table in which musical events can be stored. We can think of it as a spreadsheet with *tracks* as columns and *lines* as rows.

Musical events are defined with pure text. The event `C#3 5-32-000` plays the note C sharp at octave 3 using instrument number 5 at volume 32. The last three zeroes can be used for various types of sound effects, or to set new tempo.

The tables are called *patterns*, and a song usually contains several patterns. To control the order patterns are playbed back, we use a *playlist*. For example, if we have three patterns, a typical song could have a playlist like this: 1, 2, 1, 2, 3, 1, 2.

1.1 How Radium Differs from the Classical Tracker Interface

Radium[4] differs from the music tracker interface by using graphical elements instead of text and by allowing any number of events to be placed anywhere.[5] The latter means that a line in Radium is essentially just a graphical hint. It should be possible to compose millions of years of music within just one tracker line.

These differences are so fundamental, that it's questionable whether Radium can be defined as a tracker.

1.2 History of Radium

The first version of Radium was released in year 2000 under the GPL license, and it only supported MIDI. After the initial release, Radium was developed actively for around a year, followed by a period between 2001 and 2012 with less development. Since 2012, Radium has been actively developed again.

The features presented in this paper have mostly been implemented in 2012 and 2013. Audio support was introduced in November 2012.

[1] According to Wikipedia: `http://en.wikipedia.org/wiki/Music_tracker`

[2] `http://en.wikipedia.org/wiki/Ultimate_Soundtracker`

[3] When I started making radium, I also considered letting time go in the horizontal direction. I don't remember why I chose the vertical direction.

[4] `http://users.notam02.no/~kjetism/radium/`

[5] This feature may not be useful, depending on how you compose music. But at least when using splitted lines, and for accurately importing midi files and other music formats, it's a necessary feature.

1.3 Portability

The first version of Radium was released for the Amiga Operating System (*AmigaOS*), version 3.0 or later. The code was written in a portable style, where non-portable code was clearly separated and easy to replace. An alpha version for Linux was available already in 2001.

Radium is at the time of writing available for Linux, Windows, and Mac OS X, where Linux is the main development platform and the platform with the most features. It should be straight forward to port Radium to a platform which has Jack, POSIX, and Qt.

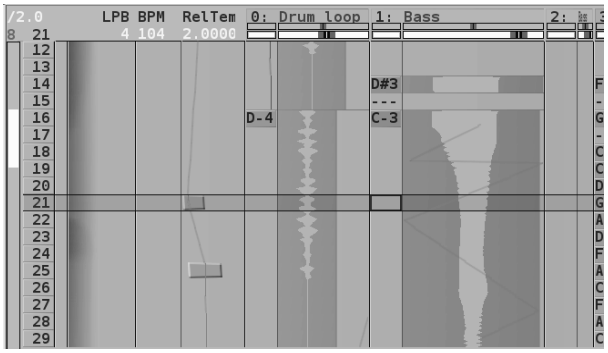
1.4 Term

In the rest of this paper, the word "line" means a tracker line, and not a vertical graphical line (i.e. a row of pixels) or an automation line. In cases where we refer to a graphical line, the expression "graphical line" will be used instead. In cases where we refer to an automation line, the expression "automation line" or "break point line" will be used.

2 Radium Features

2.1 The Editor

The image below shows a *Block* (the name of patterns in Radium).[6] From left to right, we see a vertical slider, line numbers (12-29), a green and blue area indicating tempo, an *LPB* track (Lines Per Beat), a *BPM* track (Beats Per Minute), a *RelTempo* track (for doing time-varying tempo changes), plus two sound tracks; a drum loop track and a bass track:

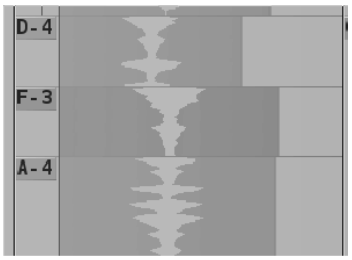


We also see that text is used to denote pitch ("D-4", "D#3", and "C-3"), while graphical break point lines are used to define tempo changes and effect automation. Pitch can be denoted with graphics too, using vertical lines, but text is clearer and more accurate.

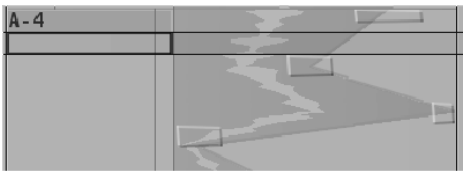
[6] The word "Block" comes from Octamed (`http://en.wikipedia.org/wiki/Octamed`). I think Block is a better name than Pattern, at least in Radium where events can be placed freely and doesn't have to follow a pattern.

2.1.1 Editor Elements

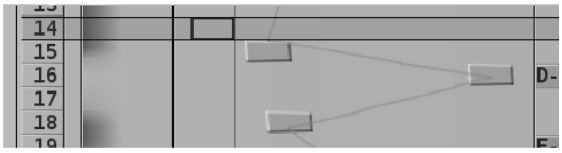
- Audio waveforms are shown in the tracks:



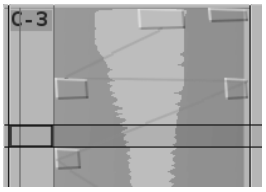
- Time-varying volume changes (crescendo/diminuendo) are defined using break point *curves*. The audio waveforms are updated in realtime:



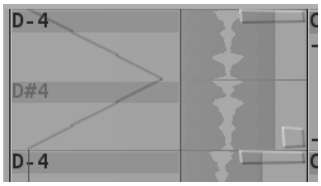
- Time-varying tempo changes (accelerando/ritardando) are defined with break point *lines*. The audio waveforms are updated in real time:



- Effects, e.g. reverb or chorus, are also defined with break point lines:



- ...And so are time-varying pitch changes (glissando's):



- Pitch can be defined with unlimited precision. The pitch below is placed 82 cents above C sharp at octave 4:

- Lines can be split. Splitting is essentially just a way to zoom in on one line so that you have more space to edit, but it can also be used to define measures. Furthermore, splitted lines can themselves be splitted, those lines again can also be splitted, and so forth:

- Updating the graphics too often can be tiring for the eyes. The SPS option (Scrolls Per Second) sets a limitation on the number of updates per second. SPS is an effective way to make the viewing more pleasurable when not using smooth scrolling.

- The velocity of new notes can be set using a random walk algorithm (drunk velocity). The algorithm tries to simulate how a musician varies volume while playing:

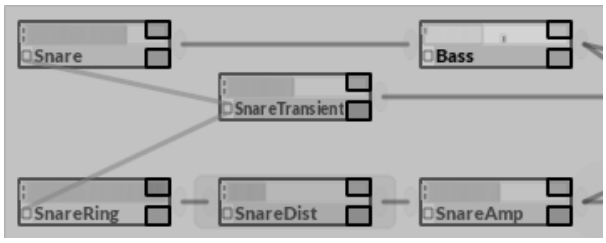
- All editing operations are undoable and redoable. The number of undoes is limited by system memory.

2.2 The Modular Mixer

The modular mixer provides a graphical interface to route note events and audio signals between sound objects.

The role of a sound object is to produce audio, receive audio, produce note events, receive note events, or any combination of those.

Inside each sound object in the Mixer GUI, there is a volume slider, a mute button, a bypass button, VU meters (one for each channel), and a "diode" that lights up when receiving note events:



The green lines show connections for note events, such as *Note On*, *Note Off*, Note volume changes, and Note pitch changes.

The other connections (those painted in a color that resembles mortar[7]) show audio connections.

2.2.1 Separate channel routing

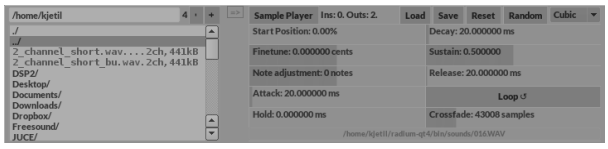
In the Mixer GUI, an audio connection sends all channels from one sound object to another. In order to (for instance) send only left channel, or only receive right channel, the audio connections must be routed through special channel routing objects.

The idea is that it's faster to use a little bit more time to route channels separately when necessary, than to always connect every channel manually.

2.3 Instruments and Audio Effects

2.3.1 Sampler Instrument

This instrument can play: 1) Normal soundfiles,[8] 2) Fasttracker instruments,[9] or 3) Soundfonts [Rossum and Joint, 1995]:



[7] "Name that color": `http://chir.ag/projects/name-that-color/#594C5B`

[8] All formats supported by libsndfile: `http://www.mega-nerd.com/libsndfile/`

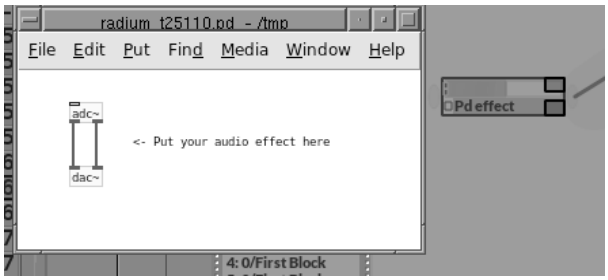
[9] Text files describing the Fasttracker "XI" instrument format: 1) "XI format description" by "KB / The Obsessed Maniacs / Reflex", 2) "The XM module format description for XM files version $0104" by "Mr.H of Triton" in 1994.

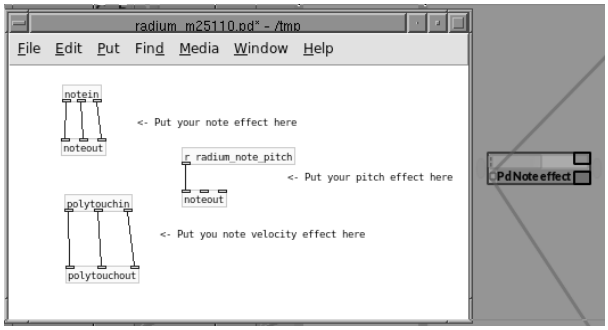
2.3.2 VST Plugins and instruments

Native VST plugins and instruments are supported on Linux, OSX and Windows:

2.3.3 Pure Data (*Pd*)

Pd processes can be inserted anywhere in the sound graph. The Pd GUI is opened by double clicking the sound object. There is no limitation on the number of simultaneous instances:





Custom Pd controllers make it possible to control Pd from Radium, and to control Radium from Pd. The Pd controllers appear as Radium effects, similarly to "Max for Live":[10]

2.3.4 STK Instruments

Radium includes 20 STK instruments doing physical modeling [Cook and Scavone, 1999]. These are written by Romain Michon in the Faust language [Michon and Smith, 2011]. Michon's instruments have been slightly modified to be used as instruments in Radium.

[10] `https://www.ableton.com/en/live/max-for-live/`

2.3.5 Zita Reverb

Fons Adriaensen's "Zita Rev1" reverb (*Zita Reverb*),[11] implemented by Julius O. Smith III in Faust [Smith, 2012].[12] Zita Reverb is also used as the default reverb when creating a new song.[13]

2.3.6 Multiband Compressor

A multiband compressor. The DSP is implemented in Faust by using components written by Julius O. Smith III [Smith, 2012].: Compressor, lookahead limiter, bandsplit, and smoothing.

2.3.7 Other Instruments and Audio Effects

a) LADSPA plugins. Richard Furse's Linux Audio Developer's Simple Plugin API. *b*) Maarten de Boer's multitap delay "Tapiir" [De Boer, 2001], implemented by Yann Orlarey in Faust. *c*) A Fluidsynth instrument, using libfluidsynth.[14] *d*) Sound objects to send or receive audio to and from jack clients. *e*) A pipe object. *f*) Channel routing objects (section 2.2.1). *g*) MIDI output.

2.4 Instrument and Plug-in Configuration Widget

Sound goes through five parts in the instrument and plugin-in configuration widget. From left to right in the picture below, we see: 1) A *Note Duplicator*, 2) An automatically created Plugin/Instrument GUI, 3) A Compressor, 4) An Equalizer, 5) Settings for dry/wet, panning, stereo width, reverb, chorus, and output volume:

1) Instruments can play several note events when a note is played, using the *note duplicator*. This is convenient to, for instance, double the bass, or add a simple echo. Up to six notes can be played for each incoming note event, and for each of those six notes, the user can specify values for transposition, volume change, delay, and duration. If

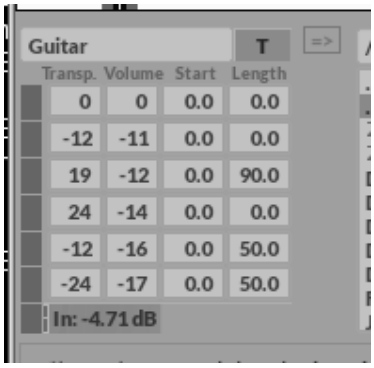
[11] `http://kokkinizita.linuxaudio.org/linuxaudio/zita-rev1-doc/quickguide.html`

[12] `https://ccrma.stanford.edu/~jos/Reverb/Zita_Rev1_Reverberator.html`

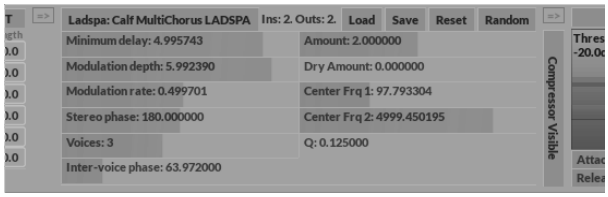
[13] The "Calf Multichorus" LADSPA plugin (written by Krzysztof Foltman) is used as the default Chorus effect.

[14] `http://www.fluidsynth.org`

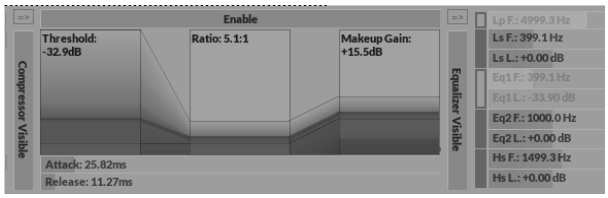
you need more than six notes, you can connect the sound object to yet another sound object to duplicate the notes further:



2) Sliders and buttons are automatically created for all instruments and plug-ins, based on the controllers they provide:

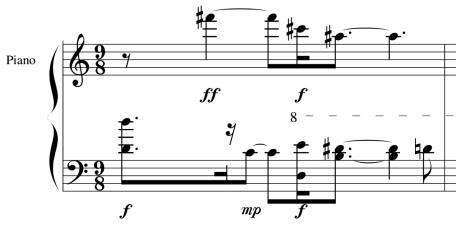


3) The compressor has a novel interface which tries to show more intuitively how the sound is squashed together. The DSP code is written in Faust by Julius O. Smith III [Smith, 2012].



2.5 Common Music Notation

Scores can be generated from Radium files automatically with Bill Schottstaedt's notation software *Common Music Notation* (CMN) [Schottstaedt, 1997].[15] The generated scores can be further tweaked in CMN, either by editing the generated CMN code, or by writing code that further modifies the CMN code. The latter technique is used to generate this score:



3 Implementation Details

Radium is mainly written in C and C++. Some code is also written in Python, Faust [Orlarey et al., 2009] and Scheme.[16]

3.1 Painting the Editor

The visible part of the editor is painted line by line to a backbuffer. When the editor is scrolling, we just copy corresponding tracker lines from the backbuffer into the screen. When a tracker line is not visible anymore, it is marked as free, and available for painting a new line. This way, we don't have to repaint everything for every update or scroll the screen or the backbuffer.

Unfortunately, this strategy causes the order of the lines in the backbuffer not to be chronological (newer lines often appear below older lines). Non-chronological order makes it impossible to paint graphical elements that span several lines in one operation. This limitation causes breakpoint boxes to be squashed up against the ceiling and floor of a tracker line,[17] and automation lines to be slightly not quite connected (or too much connected) because of anti-aliasing artifacts.[18] Scrolling the backbuffer[19] would not solve the problem either since graphical elements can start before the visible area, or end after the visible area. Therefore, at least some graphical elements has to be painted in several operations anyway.

An alternative solution that would solve the graphical problems, is to make the backbuffer big enough to contain the complete block. But this solution could occupy too much memory.[20]

[15] `https://ccrma.stanford.edu/software/cmn/`

[16] Using the Guile interpreter

[17] This effect probably looks more like a feature,

[18] while this effect probably looks more like a bug.

[19] or modifying Qt so that the underlying coordinate system would match the order of the lines in the backbuffer

[20] There are of course other solutions as well that would solve the graphical problems while still keeping the backbuffer, but I think they would complicate the code too much to be worth the effort.

However, since today's desktop computers (2014) seems fast enough to just repaint the screen when necessary, the strategy of painting tracker lines one by one in a backbuffer will, albeit so efficient that it made the program usable on hardware from 1992,[21] probably be removed in the near future. The advantage of not using a custom backbuffer is simpler code and less graphical artifacts, plus that it is simpler to add new graphical features when graphical operations are not bounded to be performed within tracker lines.

3.2 Smooth scrolling

Music trackers have traditionally updated the screen only when the current tracker line changes (i.e. scrolling line by line). By updating the screen at each vertical blank instead, we get smooth scrolling.

Smooth scrolling looks amazing compared to scrolling line by line, but perhaps more importantly is that smooth scrolling seems significantly less tiring for the eyes.

3.2.1 Render using the CPU

The first attempt to achieve smooth scrolling was to make Radium render the screen by copying line by line from the backbuffer at each vertical blank. To achieve sub-pixel accuracy, all painting operations on the backbuffer were performed n times, painting to n different back buffers, where each backbuffer was slightly skewed to the next one, all within the span of one pixel in the vertical direction. A good value for n would be at least 4.

One problem with this attempt was that the amount of time to render a frame varied a bit, and it was easy to lose the vertical blank deadline and get a frame glitch. The usual vertical blank period for an LCD screen is $16+\frac{2}{3}$ ms, so we don't have much time, and we can't trust the OS to wake us up soon enough if the current process for some reason has yielded in the middle of rendering.

A graphical glitch is very apparent when the whole screen moves in one direction at a constant speed, so to avoid frame glitches, Radium rendered frames in a separate thread and put them on a ringbuffer which the main thread would read from.[22] If a single frame took more time to render than $16+\frac{2}{3}$ ms, we still avoided a glitch if the average rendering time was less than $16+\frac{2}{3}$ ms.

However, this strategy didn't play very well with the current painting system (i.e. the code became very complicated), plus that it had a quite high CPU usage (which also made it more prone to frame glitches), so it was abandoned.

3.2.2 Render using the GPU

A more successful attempt at achieving smooth scrolling has been to use OpenGL in 2D mode. By letting the GPU repaint everything at each vertical blank, we achieve both smooth scrolling and a very low CPU usage. Another advantage is significantly smaller and simpler code since we don't use the type of backbuffer described in section 3.1 plus that scrolling is only a matter of sending updated y coordinates to OpenGL for the graphical objects.

This code is currently under development and should replace the current system soon.

3.3 Embedding Pd

Radium uses Peter Brinkmann's *libpd*[23] as basis to embed Pd.

Libpd is a thin layer of code that makes Pd into a library [Brinkmann et al., 2011]. Libpd doesn't include the Pd GUI, and it has some other limitations as well, so a "Radium fork" of libpd has been made for including features needed by Radium.[24]

The first modification was to re-add the GUI and create an API to control it. Several other enhancements and required modifications followed, such as loading and saving patches and adding a *void∗* argument to the midi functions.

3.3.1 Libpds (libpd with an extra 's')

However, the biggest challenge for using libpd is that only one Pd instance can run in a process simultaneously. With only one instance, you can't send sound from one patch to another in the Radium mixer (at least not if there is a non-pd sound object in the middle of those two). Or, for that matter, you can't make a LADSPA or VST plugin out of a Pd patch.

To circumvent this limitation, an additional library called *libpds* has been added to the Radium fork of libpd. Libpds makes it possible to load several Pd instances and communicate with them separately. Libpds has almost the same API as libpd, except that most functions take an additional "pd instance" parameter.

[21] Amiga 1200

[22] This strategy is similar to how we reliably get sound in real time from a non-deterministic source, for instance a hard drive.

[23] `http://libpd.cc`

[24] `http://github.com/kmatheussen/libpd`

Libpds works by dynamically loading a new libpd library file for each new Pd instance. To avoid symbol clash for the global variables between the various Pd instances, *dlopen* is called with the *RTLD_LOCAL* flag when opening "libpd.so". The RTLD_LOCAL flag prevents symbols from being shared globally.

Unfortunately, this behavior causes problems when loading Pd externals (i.e. plugins which are loaded during runtime). Pd externals require access to functions and global variables provided by Pd, but since Pd doesn't share its symbols globally, the externals fail to load.

The selected solution for the problem is to statically link the most common Pd externals into libpd. 921 externals are currently included, and among them are most of the externals distributed with the Pd distribution *Pd-Extended*.[25] In order to compile that many externals without manually writing a large Makefile, a script recursively scans a list of directories and compiles all externals it can find.

A slightly simpler way to load externals would be to link the Pd externals directly (i.e. instead of recompiling), but using a ".so" file as a static library does not work.

It is likely that there are better ways to support externals, such as implementing a new dynamic linking system, but the current solution seems to work well for now

3.4 Garbage collection

Radium has from the start used Hans Boehm's garbage collector for C and C++ as memory manager (BDW-GC) [Boehm and Weiser, 1988]. It is not necessary to free memory manually when using a garbage collector, so Radium has fewer lines of code, and most likely fewer bugs, because of this choice.

There has been no trouble with BDW-GC, and Radium has not had memory leaks. It is strange that BDW-GC is not used in most large programs written for C or C++.

4 Related software and how their features compare to Radium

4.1 Jeskola Buzz

Jeskola Buzz[26] appeared in 1997-1998.[27] Jeskola Buzz was probably the first tracker with a modular mixer. The modular mixer in Radium is inspired by the one in Jeskola Buzz, but the modular mixer in Jeskola Buzz doesn't support sending note events or sound objects with more than two channels.

4.2 Aodix

Aodix[28] was released before 2002, but I don't know when. Aodix may have been the first *tickless* tracker, depending on how old Aodix is. Tickless means that events are not bounded by tracker lines, a feature which is shared with Radium. Another feature shared with Radium is that you can apparently zoom in and out of the patterns.

4.3 Renoise

Renoise[29] was released in 2002. Renoise is a more traditional tracker than Jeskola Buzz and Aodix, but has more features.

Renoise uses one instrument per track, which is similar to Radium, but Renoise lets you organize tracks further by optionally grouping tracks and instruments. For instance by grouping all drum tracks or all vocal tracks. Grouping makes patterns visually clearer and simpler to navigate and it simplifies adding effects to a group of instruments (since they are already grouped). Grouping is a feature that is currently missing in Radium.

Renoise also supports effect automation and tempo automation, but unlike Radium, the graphics is placed horizontally in a separate area below the tracks, and not in the tracks themselves.

5 Conclusion

Radium presented a radical change to the classical tracker interface when it was released fourteen years ago.

The following is a list of larger tracker features that first appeared in Radium (at least to my knowledge). An appending * means that Radium is still the only tracker, or tracker-like, program that provides this feature, at least to my knowledge:

a) Smooth scrolling*; *b*) Limitation on the number of scrolls per second*; *c*) Tickless timing (may have been introduced in Aodix before Radium); *d*) Zoom in/out (may have been introduced in Aodix before Radium); *e*) Waveform data visible in tracks; *f*) The "Radium Compressor" compressor interface*; *g*) Pitch values shown graphically; *h*) Tempo automation*; *i*) Effect automation*; *j*) Volume automation*;

[25] http://puredata.info/downloads/pd-extended
[26] http://www.jeskola.net/buzz/
[27] http://en.wikipedia.org/wiki/Jeskola_Buzz
[28] http://www.kvraudio.com/product/aodix-by-arguru-software/details
[29] http://www.renoise.com

k) Pitch automation*; *l*) Adjustable track widths*; *m*) Pd or Max/MSP integration*; *n*) Track headers with volume control and instrument name*; *o*) Automatic MIDI preset change when playing note for instrument with different preset*; *p*) Line splitting (including line split splitting, line split split splitting, etc.)*; *q*) Unlimited number of simultaneously playing notes per track, and no limitation when they are allowed to start and stop playing*;[30] *r*) Unlimited number of blocks, tracks and lines; *s*) Generate scores with CMN*; *t*) Unlimited undo/redo; *u*) Send pitch change events between instruments*;[31] *v*) Configurable menus*.

This list of (more or less useful) new features shows that Radium has tried to be an innovator for tracker software. Radium will try to be an innovator in the future as well.

6 Acknowledgements

Radium is an open source program which includes code from several other programs and uses several open source libraries. Today's Radium would not exists without the open source community. Some of the people who have written code that's used in Radium are (apologies to those I've forgotten):

Fons Adriaensen: Zita REV1; Conrad Berhörster / Josh Green / Peter Hanappe / David Henningsson / Pedro López-Cabanillas / Antoine Schmitt: Fluidsynth; Michele Bosi: Visualisation Library; Hans Boehm / Ivan Maidanski: BDW-GC; Peter Brinkmann: libpd; Rui Nuno Capela: code from QTractor to auto-create Plugin GUI's and show VST GUI's; Paul Davis / Stephane Letz: Jack; Ray Donnelly / Alexey Pavlov / Roumen Petrov: MinGW Python; Dominique Fober / Albert Gräf / Stephane Letz / Yann Orlarey / Julius O. Smith III: Faust; Krzysztof Foltman: The CALF multichorus LADSPA plugin; Grigor Iliev: The Soundfont parser in libgig; Giles Hall: The python-midi library; Bob Ham: Code from Jack-Rack to organize LADSPA plugins using liblrdf; Steve Harris: liblrdf; Erik de Castro Lopo: libsamplerate and libsndfile; Romain Michon: The Faust STK instruments; Paul Mineiro: Fast functions to calculate exponential and logarithmic values; Javier Serrano Polo: Vestige; Miller Puckette: Pd; Yann Orlarey: The Tapiir effect implementation and smooth delay code; Bjorn Roche: Memory barrier code; Gary P. Scavone: RtMidi; Bill Schottstaedt: CMN; Julius O. Smith III: Compressors / lookahead limiter / filters / equalizer; Hans-Christoph Steiner et al.: Pd-Extended; www.magnetophon.nl: The included Blowfish demo song; TumaGonx Zakkum: LADSPA plugins for Windows.

I also want to especially thank Yann Orlarey for creating the Faust programming language and Julius O. Smith III for all the DSP code he has written for Faust. Their work has saved me a lot of time and ensured professional sound quality.

[31] I.e polyhponic aftertouch for pitch instead of volume

References

Hans-Juergen Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820.

Peter Brinkmann, Peter Kirn, Richard Lawler, Chris McCormick, Martin Roth, and Hans-Christoph Steiner. 2011. Embedding Pure Data with libpd. In *Proceedings of the Pure Data Convention*.

Perry R Cook and Gary Scavone. 1999. The Synthesis Toolkit (STK). In *Proceedings of the International Computer Music Conference*, pages 164–166.

Maarten De Boer. 2001. Tapiir, a Software Multitap Delay. In *Conference on Digital Audio Effects, Limerick, Ireland*.

Romain Michon and Julius O Smith. 2011. Faust-STK: a Set of Linear and Nonlinear Physical Models for the Faust Programming Language. In *Proceedings of the 11th International Conference on Digital Audio Effects (DAFx-11)*, page 199.

Yann Orlarey, Dominique Fober, and Stephane Letz. 2009. FAUST: an efficient functional approach to DSP programming. *New Computational Paradigms for Computer Music, Editions Delatour France*, pages 65–96.

Dave Rossum and E Joint. 1995. The SoundFont® 2.0 File Format.

Bill Schottstaedt. 1997. Beyond midi. chapter Common Music Notation, pages 217–221. MIT Press, Cambridge, MA, USA.

Julius O Smith. 2012. Signal Processing Libraries for Faust. In *Proceedings of the Linux Audio Conference 2012*, pages 153–161.

Music and Art Programme

Klangdom Concert I

Thursday, 01.05.2014, 20:00 h, ZKM_Cube

Anthony Di Furia: Through the Space of Crying

The project is based on tin (the chemical element) as a conceptual starting material and the analysis of the "tin cry", the characteristic sound a bar of tin makes when bent at room temperature. TIN META-SONIFICATION SYNTH is a software written in SuperCollider based on two sonification processes: the first is derived from the physical-chemical characteristics of tin, the second on the atomic number and atomic radius. The variation of pressure and temperature controls in real time the values of density, sound velocity, state of matter, boiling point and melting, which controls a first synth.

The atomic radius and atomic number are the basis of an additive synthesis complex, modulated in frequency, amplitude and phase. The generated sound is spatialized with first-order ambisonics (ATK Ambisonic-Toolkit), according to the theory of atomic orbitals.

The composition/improvisation derivative is a journey through the sonic dimensions of tin, it creates a "bond" between the real sound of the tin cry and an imaginary soundscape.

Patrick Hartono: The Complete Series of Kecapi (2012–2013)

The Complete Series of Kecapi (2012–2013) is a merger of three different compositions (*Kecapi I, II, III*) that I did for Gaudeamus Muziekdag (Gaudeamus Jonge Componistenbal) at Rasa theatre Utrecht in January 2014.

Kecapi is a series of electroacoustic compositions that I started in October 2012, and that has been developed into four different versions. The Sound Material of *Kecapi* is basically manipulated recorded sound of Kecapi/Sitar (an Indonesian traditional plucked string instrument) that was recorded in Jogjakarta. Each individual version of *Kecapi* has a different approach and concept.

Kecapi I was selected to be premiered on Sound Gallery During Wocmat 2012, Taiwan. *Kecapi II* was selected to be premiered during the WOCMAT 2013 concert, and also selected as finalist of the Taiwan International Electroacoustic Music Award. *Kecapi III* was premiered as part of the Behind The Score Concert at Codarts Rotterdam Conservatorium 2013.

José Rafael Subía Valdez: Chiral

Inspired by the chemical sense of the word, *Chiral* is a piece that tries to apply some of this property of chemical symmetry to music. It seemed interesting to write a piece after thinking about the chirality of our hands when playing the piano: some chords can only be played by a particular hand, left or right. Furthermore, the relationship between the piano and the computer-generated sounds is always arranged symmetrically in time, gesture or pitch. Conceptually, symmetry is as well considered in its form and sonority.

Chiral uses a Pure Data patch to analyze, process and generate sounds; all audio is produced in real time. It was composed using the PCSlib during two stages of production; a Computer Assisted Composition stage and the programing of the performance patch itself.

Giorgio Klauer: Haar

Decomposing sound into particles, sensitivity and masking effects in auditory perception, friction in bowed instruments are the themes getting intertwined in this composition and signed in the title: *Haar* (hair), like Pferdehaar, Haarzelle, Alfréd Haar.

The sound actuation model is the bowed instrument's, yet it has been implemented through 50–70 cm long, thick, black human hairs gently rubbed against a moving magnet phono cartridge cantilever. The sonic characterization has been afterwards dramatically emphasized by means of a granular composition environment programmed in sclang. In this implementation, envelope, pitch, spatialization, indexing and further grain controls have been imposed by perceptual feature descriptors extracted by the very same sounds, with the result of an anamorphic and multidimensional microediting process.

Louise Harris: sys_m1

sys_m1 is an eight-minute electroacoustic composition realized using *systemic*, a system I constructed for real-time composition, performance and sound spatialisation controlled via a physics-based visual environment. In systemic, physics-based algorithms govern the behaviour of objects in a visual system, and the movement of those visual objects controls the spatialisation, via vector base amplitude panning, of corresponding sound objects over an 8-channel circular speaker configuration. *sys_m1* is composed from a number of recordings taken from systemic. The sonic material is a combination of pre-composed sound objects and real-time synthesized sound. By utilizing a physics-based visual system to control the spatialisation of sound, I am effectively removing decision-making from the spatialisation process.

Martin Hünninger: Spaces

Spaces explores the relation of two different sound objects in three different spaces. The inspiration to this piece stems from the mathematical concept of topological space, a structure that allows one to define notions such as connectedness, continuity, inside, outside, openness and closedness. The composition develops a path from the pure abstract space through the inside of an acoustically closed room into an outside scenery. The composition is written completely in SuperCollider and runs on any up to date computer. It features aleatoric elements, thus each performance differs in a subtle way from the other, while the overall structure is fixed.

Klangdom Concert II

Friday, 02.05.2014, 20:00 h, ZKM_Cube
Klangdom concert with productions of the ZKM | Institute for Music and Acoustics

Concert "Playroom"

Friday, 02.05.2014, 22:00 h, ZKM_Music Balcony

Jürgen Reuter: Random Noise: Concert for Sound Column Four Hands

Two players give a concert in a competitive manner. They put up and rearrange colored shapes and symbols on an advertising column that slowly rotates. The surface of the column is scanned, and a computer program renders the shapes and symbols into sound, as they move under a virtual play head cursor that is projected onto the column. Since the players compete in an uncoordinated fashion rather than cooperate, the overall picture grows wild. Both players are struggling to dominate the system by putting as much information as possible onto the column. As their competition finally results in a big chaos, the overall informational content approaches zero, resulting in random noise.

Bruno Ruviaro & Carr Wilkerson: Vowelscape 1.0

Vowelscape 1.0 is a collaborative audiovisual performance by Bruno Ruviaro (Santa Clara University) and Carr Wilkerson (CCRMA/Stanford). Strangled robotic voices and flickering letters are some of the building blocks of this study on the poetic resonances of isolated vowels.

Mauricio Valdés & Juri Pohleven: DNA Sequencer

DNA Sequencer is an ongoing project based in Slovenia dedicated to create artistic outputs involving realtime sound art generation, video interaction and genetics. Our research is spreading continuously in order to create complex networks of information that can be used for our performances. The project is conceived as an inter-media show performance presentation in one act without any cuts, consisting of realtime generated sound art and video controlled with Pure Data and Max/MSP, all supported by one synthesizer, one prepared electric guitar and one laptop computer.

The performance begins with a sequence of notes with fully mechanized sonic and visual events without human control; events are an exact replica of the translation of the genetic code into music and image. Subsequently a certain signal on a DNA sequence triggers gradual incorporation of realtime sound events through which the performers also reversely affect the visualization encoded by the genome. This starts some sort of a dialogue/battle between the concepts arising from the genome's output and/vs. the performers' improvisation. Accordingly, the piece is a hybrid of improvisation and sequencing, which imitates the fluid environment in which the rigid genome exists and reflects the epigenetic view of biological systems. Inside an ideal scenario, the events that occur live overlap with the sequenced, leaving the resultant sound and image as a live expressiveness. The genetic algorithm is eventually dominated and subjected to

control parameters that keep the work performance running; it's participation and control remains essential for parameterizing certain events of our electronic instruments, but without its own sound. The play ends as the musicians decide on stage ...

Bruno Ruviaro & Juan-Pablo Caceres: Panela de Pressão

Panela de Pressão is an improvisation over the network with Bruno Ruviaro and Juan-Pablo Caceres. Juan-Pablo will be playing live from Santiago, Chile. The two musicians started playing together in 2004 when they first met in the United States. After a long hiatus, the duo finally resumed playing last year, now mostly through network performances using JackTrip. "Panela de Pressão" means "pressure cooker" in Portuguese.

Louise Harris: intervention:coaction:

The project is a live, audiovisual, beat-and-noise-based performance work. The intention is to create a symbiotic system, in which live decision making by the performer impacts on both the audio and visual components of the work but also in which both the audio and visual components can interact with one another, causing behaviours that are not directly controlled by the system performer. There is also an element of chaotic behaviour built into the system, causing unpredictable audio and visual outcomes.

Malte Steiner: Elektronengehirn – Concert reqPZ

Audiovisual electroacoustic concert by Malte Steiners project *Elektronengehirn*. The input of piezo contact microphones are analyzed with Pure Data on a Linux laptop controlling sound and graphics. The usage is between percussion trigger and pick up, sometimes the piezo sound is used directly, in other parts only as control data for synthesis and the visuals. The contact mics are attached to a metal plate which is played by strokes and beats with sticks, realizing an expressive performance.

Klangdom Concert III

Saturday, 03.05.2014, 20:00 h, ZKM_Cube

Luis Valdivia: Xaev1uox

The piece was finalized om March 2014, and was realized using SuperCollider on Fedora 19/20.

Florian Hartlieb: Out of the Fridge

Out of the Fridge was composed as a ballet-insertion for a new staging of Christoph-Willibald Gluck's opera "Il Parnaso Confuso", which was premiered in the Schönbrunner Schlosstheater in Vienna in 2011. Alienated sounds from a refrigerator, like the fridge buzzing, shaking ice cubes or the clicking noise inside the freezer are mainly the source material for the piece (in the ballet, the refrigerator was an important part of the scenery). All sound processing was realised with the language Csound. The first part of the work is about constructing and deconstructing, with a clear harmonic structure

and some rhythmic elements. The second part dissolves the harmonic structure, it has more confusion and a processed collocation of the material.

Fernando Lopez-Lezcano: Divertimento de Cocina

Divertimento de Cocina (Kitchen Divertimento) stages several kitchen scenes, with sounds and rhythms layered, controlled and triggered by a live performer. Kitchen utensils are mixed, transformed and orchestrated in real-time through a LaunchPad controller and a custom set of SuperCollider classes and programs. What are initially extremely simple rhythms get progressively more complicated as they are layered together in increasingly thicker textures in the initial section of the piece. While the performer walks you through different soundscapes, the initial rhythms form the backbone and guide for the rest of the piece. The SuperCollider program also spatializes all sounds under the control of the performer in a 3D soundscape that can be diffused through an arbitrary number of speakers (the original sound stream is internally generated in Ambisonics, with at least 3rd order periphonic resolution).

Clemens von Reusner: rooms without walls

rooms without walls has been composed in 2012 for an 4x4 array of loudspeakers built at the Platz der Weltausstellung (Expo 2000) in Hanover, Germany. The arrangement of the 16 speakers/light steles in the sculptural appearance is strictly geometric. In an abstract way it reminds of geometric spatial divisions in baroque gardens as the can still be found in the Hanover Royal Gardens. In the composition each four corners of a square define 14 square and overlapping areas of different size and position. Hence 14 virtual rooms are implemented using Ambisonics as sonic rooms to be equipped with different sounds that in each room are moved in individual orbits simultaneously.

The sound material on which this composition is based has been designed in terms of its spectro-morphological development and its structure contrasting with the existing sound of the public space. The third-order ambisonic spatialization was done with Csound. Due to the very unique original setting of the 4x4 loudspeakers, an 8-channel concert-version of the piece is played.

Ali Ostovar: A Thin Light Behind the Fog

The title of this piece is very descriptive but it is more about sounds, based on spectrum and timbre. Physical modeling synthesis, granular synthesis and some other techniques were utilized.

Bernardo Barros & Mário del Nunzio: Improvisation

Sound Night

Saturday, 03.05.2014, 22:00 h, ZKM_Music Balcony

Wolfgang Spahn: ENTROPIE

ENTROPIE is a noise and projection performance. Both, sound and projection are based on different analogue and digital machines developed by the artist. Each system generates simultaneously structured noise as well as abstract light patterns.

The invention of moving pictures went along with an artificial separation of sound and image. *ENTROPIE* merges them again. It makes the data stream of a digital projector audible and gives an audio-visual presentation of the electromagnetic fields of coils and motors.

All hard- and software was developed by the artist as an open hard-/software system (`http://www.dernulleffekt.de`). The two cameras and the controlling system run on three Raspberry Pi. A Pure Data patch handles the controlling and a Python program manages the camera output.

Renick Bell: Algorave Improvisation

This performance of improvised programming generates *algorave*, danceable percussive music emphasizing generative rhythms. The rapidly changing algorithmic bass music is intended to stimulate dancing. Using a custom live coding system called Conductive with the Vim text editor and GHCi, the Haskell language interpreter, multiple concurrent processes are used to trigger a SuperCollider-based software sampler loaded with thousands of audio samples. At least two methods of rhythm pattern generation are employed: stochastic methods and L-systems. Patterns from both are then processed to generate variations with higher and lower density, which are deliberately chosen during the performance. The performance also involves programming to control other parameters. The programming activity is projected for the audience to see.

Tiny Boats (Jason Jones & Jesse Crowley): Burn in the Sun

A song composed by two. Recorded, edited, mixed, and mastered in Linux with Harrison Mixbus at Art City Sound in Springville, UT.

This was the last song we produced for our album 'The Broken Vessels'. It was created specifically to be the first song on the album to set the tone and establish some themes and expectations for the rest of the songs. The song itself is a journey from a lost, cynical world view into a triumph of thought and hope. It helps to illustrate that the interactions we have with people help to shape our way of seeing the world, and better understand our place in it.

Yan Michalevsky: Locum Meum

Locum Meum is an electronic composition using synthetic sounds and produced using MuLab DAW. Nevertheless, some melodic parts can be optionally played live and were composed with the real instrument constraints in mind. While aiming at the House/Techno genre, this composition is very melodic, subjective and manifests the personal musical preferences of the composer. Synthetic sound combined with a classical violin part, and rock drum fills, all of these try to to blur the genre boundaries.

Unsound Scientist (Amos Przekaza): Selected Works DJ Set

Selected Works is a collection of pieces composed mostly during 2013. Consisting of about 10 songs, it represents the learning process of using Linux tools for music production and compositional exploration in general. All works were made entirely on with Linux software, mainly LMMS and Ardour along with ZynaddsubFX, AMsynth, Hydrogen, TAL noisemaker, Qtractor, Phasex, Synthv1 and other tools packaged with the KXstudio distribution. Live instrumentation along with software and hardware synthesizers were used as well.

Yen Tzu Chang: Self-Luminous 2 – Unbalance

Self-Luminous 2 is a little bit different from first one: It is much easier to control, not heavy, and the design is more organized. I also added a HMC6343, an electronic compass sensor for the Arduino. When I move the instrument to different directions, some accidental sound will be effected (e.g., some sound suddenly disappear or are covered by other sounds). Sometimes, this is a bit risky ... A part of my performance is impromptu. When I play, it is interesting that the performance is severed violently from the âĂIJselfâĂİ. The instrument was built using Pure Data and Arduino.

Vincent Rateau & Daniel Fritzsche: Superdirt^2

Superdirt^2 – fascinating electro beats mixed in with virtuously performed cello sounds, which give a result of a never before achieved danceability! With Ras Tilo at the synthesizers and Käpt'n Dirt with the cello it provides a musical experience which is situated between drum'n'bass, jungle, dub, dubstep and even far beyond ...

Jeremy Jongepier: The Infinite Repeat

A musician with over 20 years of experience and a computer with Linux. That's what it boils down to. The result: conventional, decent song-writing, with an eclectic tinge because of the choice to not walk the threaded paths coupled with an auto-didactic background, an outspoken personal taste and an open-minded world-view. The keyword for this year's Linux Sound Night is danceability. So that is what The Infinite Repeat will be bringing on stage, danceable tracks in the best indie-electronic tradition with an emphasis on melody and finding the right balance between traditional instruments and sounds generated by software running on the OS that has given name to this event.

Bart Brouns: The WOP machine

One man, 160 oscillators. An improvised live performance with a realtime synthesizer controlled by singing and beat-boxing. *The WOP machine* is the subject of a workshop on May 1st, 14:45 h in the Workshop-space.

William Light: visinin – Modern Electronic Club Music

Kentucky-born and Ohio-raised, William Light now lives in Germany, writing software and music. With his solo project *visinin*, he explores the sounds of modern electronic music, everything from heavy club beats to relaxed, downtempo soundscapes.

Jakub Pišek: Turbosampler

The audio-visual aspect of the performance absolutely refutes the illusion that the world is showing us a legible face that we need only decipher. This is a game of provisional meanings that do not lead anywhere. Synchronized A/V samples are handled not like pieces of music, but like potatoes. Everything is controlled by controllers everybody knows, and probably owns. Keyboard and mouse are on the table, just the screen has a different meaning here.

Music and Sound Art Installations

Thursday–Saturday, 01.–03.05.2014, 14:00–18:00 h

Louise Harris: Indusium

Indusium is a work-in-progress; an exploration of additive compositional process. Groups of material are added to and extended step by step, the visuals reflecting the changing timbral colours caused through unexpected crossovers in pitch and rhythm.

Lightune.G (Miodrag Gladović & Bojan Gagić): Lighterature Reading: Chapter 12

Lighterature Reading is an ambient audio/visual luminoacoustic installation. Chapter 12 consists of nine solar panels that convert light and video projection into sound images. The composition is seventeen minutes long and is repeated in a loop. Its duration was chosen as the ideal length of one side of a vinyl LP record, twelve inch. Besides the basic composition, which is fully programmed by authors, it also includes the audience intervention with different types of hand lamps, which they can pick at the entrance. Each visitor who wishes to intervene in the work can also enter a personal email address on the computer near the entrance to get a snapshot of the composition with his or her intervention as an audio recording via email. The duration of a snapshot is three minutes, which is the ideal duration of one side of a vinyl single, seven inch.

Hanspeter Portner: CHIMAERA – The Poly-Magneto-Phonic Theremin

The Chimaera is a touch-less, expressive, network-ready, polyphonic music controller released as open source hardware. It is a mixed analog/digital offspring of the theremin: An array of analog, linear hall-effect sensors and their vicinity make up a continuous two dimensional interaction space. The sensors are excited with neodymium magnets worn on fingers. The device continuously tracks and interpolates position and vicinity of multiple present magnets along the sensor array to produce corresponding low-latency event signals. Those are encoded as Open Sound Control bundles, transmitted via UDP/TCP to a Linux host running a dynamic event dispatcher, translated to musical events and finally rendered to audio via SuperCollider, Yoshimi and amSynth according to ever morphing mappings in correspondence to the visitors input dynamics. Visitors are free to interact with and experience the expressiveness of the continuous-pitch instrument.

Wolfgang Spahn: Bild einer Ausstellung. An Installation of Audio/Visual Interferences

100 years ago the Futurist Luigi Russolo introduced the Intonarumori and Bauhaus Artist Lázló Moholy-Nagy built his "Light-Space Modulator". The first apparatus generates noise and the second generates a moving light/shade pattern. Both aspects were part and parcel of the installation *Bild einer Ausstellung*. A laboratory-like setting functions as the generator of both abstract projections and corresponding noise. In contrast to the composition of Modest Mussorgski "Pictures at an exhibition" the sound is directly recorded from the picture.

The "Picture Disc" is a transparent miniature picture based on chemical-magnetic experiment with liquid iron. A laser scans optically the rotating picture. After transforming the light beam into an electric signal it is modulated by an electric circuit based on an old analog 808 drum machine. A Raspberry Pi camera with a mounted macro lense films and magnifies the abstract picture and projects it above the whole installation. All technology and machines are based on open hard- and software and documented on the artist's web side `http:/www.dernulleffect.de`.